RENAR

Standard + Guide + Reference + author meta-documents (full archive)

RENAR · Full Archive · Version 1.0 | 17.07.2026

Authors: Vadim Soglaev, Andrey Yumashev

RENAR Core

Version: 1.0 · **Date:** 2026-07-14 · **Site:** renar.tech

> ***What this is.*** *A conceptual overview of RENAR for the human reader: what the standard is about, why it is needed, and how it works at the top level. Without technical detail, frontmatter, the lifecycle, or normative rules — that is the domain of the full RENAR Standard.*
>
> ***Reading time:*** *≤ 10 minutes.* ***Who it is for:*** *PMs, lawyers, regulators, and engineers encountering RENAR for the first time. If you are an AI agent, read the Standard directly — you do not need Core.*

What RENAR Is

RENAR (*Requirements Engineering & Normative Adaptive Regulation*) is a normative requirements-engineering standard for development with AI agents. The standard governs:

- **The data model** of requirements artifacts: BR (business requirement), SR (system requirement), TR (task), ADAPT (the interpretation of the TZ), ACTZ (the TZ clarification protocol), 11 SPEC types (architecture, API, data, integration, process, UI, AI, security, operations, test environments, delivered documentation), TC (test cases), and AT (acceptance tests derived from the contract).
- **The lifecycle and Quality Gates** (QG-0..QG-4) — artifact states and the conditions for transitions.
- **Substrate capabilities** V1–V6, which the artifact storage system MUST satisfy: immutable history, atomic changes, comparison/review, branching, end-to-end version pinning, author + timestamp.
- **Conformance** — the RENAR-1..RENAR-5 levels, mandatory clauses, the manifest, and assessment procedures.

RENAR is a **specialization of SENAR** (the methodological foundation for development with AI agents) in the area of requirements engineering. A RENAR-conformant implementation is always compatible with SENAR; the reverse does not hold.

Why RENAR Exists

In development with AI agents, requirements live simultaneously across several artifacts: the client's contractual TZ, the engineering BR/SR/SPEC, test cases, the task description, the implementation in code. All of this is written and edited by a mix of humans and AI agents. Without formal contracts between artifacts, **requirements drift** arises — a divergence between what has been recorded, what is verified, and what is actually implemented.

RENAR closes eight normalized classes of drift:

1. **Schema drift** — artifact fields diverge between projects.
2. **Lifecycle drift** — statuses (`draft` / `approved` / `verified`) mean different things to different authors.
3. **Source-of-Truth drift** — one entity is edited in several places at once.
4. **Implementation drift** — code implements a requirement that has already been deleted or renamed.

5. **Terminological drift** — terms mean different things to different people.
6. **Order / provenance drift** — a delta-TZ is applied out of order, or references a non-existent requirement.
7. **TC ↔ requirement provenance drift** — a test verifies obsolete behavior.
8. **Test-fitting drift** — an AI agent weakens a test's criteria instead of fixing the code.

All eight classes are **structural**: they arise from the very fact that an artifact is co-owned by several authors, not from lapses in discipline. They can be closed only normatively — by recording a contract for how artifacts are linked, who writes which field, and which preconditions MUST hold at state transitions.

How RENAR Works (Conceptually)

The full path of a single requirement — from contract to acceptance:

```
flowchart TD
    K[Client] --> TZ[TZ — contract, immutable]
    TZ --> AR{Adversarial review}
    AR -->|«findings present»| AD["ADAPT — the interpretation:<br/>how engineers
read the TZ;<br/>architect's signature"]
    AR -->|«no findings»| ART["BR / SR / SPEC / TR / TC<br/>(RENAR description)"]
    AD -->|question to the client| ACTZ["ACTZ — the «TZ Clarification Protocol»:
<br/>decisions in the language of commitments;<br/>client and vendor signatures"]
    ACTZ --> AD
    AD --> ART
    ART --> IMPL[Implementation code]
    TZ --> EFF[Effective TZ = TZ + signed protocols]
    ACTZ --> EFF
    EFF --> AT["AT — acceptance tests<br/>derived from the contract"]
    IMPL --> AT
```

Key properties:

- **The TZ is a contractual, immutable artifact.** Once signed by the client, it is not edited. Scope changes are formalized through a delta-TZ; clarifications through protocols (see below).
- **The RENAR description is the Source of Truth about the system's behavior.** Code is a derived implementation artifact, not the authoritative definition of behavior. If the code does X but the SR says Y, that is a defect in the code, not "the actual requirement has changed."
- **Adversarial review.** A separate AI agent on a different model deliberately looks for what the primary agent missed: unasked questions to the client, weakened test criteria, hidden assumptions. This is a compensating mechanism against self-consistent but semantically incorrect AI outputs.
- **Positive/negative pairing of tests.** Every verifiable assertion of a requirement has at least one positive and one negative test case. AI agents readily cover the happy path and skip negative scenarios — RENAR makes pairing normative.
- **The test is not written by whoever writes the code, and a test MUST have been red at least once.** A test composed alongside the implementation checks what was written, not what was required. And a test that has never failed has proved nothing — it may simply be empty.

Two Documents: Interpretation and Commitment

The most common mistake in contract development is to merge into one document **how the engineers understood the TZ** and **what was agreed with the client**. The merger looks economical, but it breaks both functions at once: the client signs an engineering text they never read and cannot assess, while the engineer is afraid to write down an honest reading, because it will be carried off for signature.

RENAR separates these two documents, and the boundary is drawn **by audience**, not by content:

> ***Everything shown to the client and approved is a commitment. Everything not shown is interpretation.***

	ADAPT — interpretation	**ACTZ — commitment**
What is inside	How the TZ was read: translation into engineering language, term mapping, completed scenarios, the gaps and contradictions found	The questions and decisions put to the client. In the language of commitments: "the button is called this", "the deadline is that"
Who reads it	The engineer, the AI agent, the reviewer, the verifier	The client and the parties to the contract
Who signs it	Only the architect on the vendor's side	**Both parties**: the client and the vendor
Weight	An internal working document	**Contractual**

ACTZ is precisely the "**TZ Clarification Protocol No. N**" that already exists in contract practice. RENAR merely makes it the mandatory home for the client's decisions, and links the two: every finding in the interpretation that required the client's word MUST cite the clause of a **signed** protocol in which that word is recorded. No reading may rest on a decision the client never made; and no signed decision may go unreflected in the requirements.

The gain is simple and machine-checkable: the argument "is this a clarification or already a scope change?" disappears. The question is now binary — **was the decision put to the client, and is it signed?**

Acceptance — Against the Contract, Not the Interpretation

From this follows the **effective TZ**:

> ***Effective TZ = the initial TZ (with its annexes) + all signed clarification protocols.***

That is the benchmark for delivery and acceptance. The internal interpretation is **not part of it**: the client answers only for what they signed and understood. Acceptance becomes legally clean.

But there is an engineering consequence too, and it is the whole point. Ordinary tests are derived from requirements, and requirements from the interpretation. So if the interpretation is wrong, every test may be green: the system flawlessly matches a **wrong** reading of the TZ — and fails acceptance at the client. To check a system with the same understanding that built it is to guarantee that the error of understanding stays invisible.

RENAR therefore introduces a second, independent layer of checking — **acceptance tests (AT)**:

- they are written by an **isolated** agent given **only the effective TZ** as input: no interpretation, no requirements, no specifications, no code;

- that agent's model differs from the primary agent's — otherwise it reproduces the same reading, and acceptance degenerates into self-confirmation;
- before every trial run the tests are regenerated from the **current** revision of the effective TZ: otherwise a long engagement is delivered against a year-old contract;
- the product is not presented for delivery until all acceptance tests are green.

A divergence between the two layers is not noise but a diagnosis. **An acceptance test red while the ordinary tests are green is an error of interpretation**: the system is built correctly, but it is the wrong system. No quantity of ordinary tests catches that defect — which is the entire reason for the second layer.

The full lifecycle is governed through **Quality Gates**: QG-0 (approval), QG-1 (implementation), QG-2 (verification) are mandatory; QG-3 (architecture) and QG-4 (business outcome) are optional. Separate from them stands the release acceptance gate: conformance to the contract is proven before the product is presented.

The Default Executor — the AI Agent

RENAR artifacts are **by default** created and maintained by an AI agent on assignment from an engineer. The human acts as verifier and approver: reviewing the result, clarifying the task when needed, and approving lifecycle transitions.

Two things follow from this positioning that are unfamiliar when reading the standard for the first time:

- **Artifacts look dense** (dozens of frontmatter fields, lifecycle transitions, graph links) — because the primary reader is machine. The density is not bureaucracy but a requirement for "code in natural language" that the AI agent executes in subsequent steps.
- **The process overhead of maintenance is machine, not human.** An AI agent does not tire of filling in frontmatter; the volume of work is linear. To a human this overhead seems unbearable — but it is precisely this overhead that need not be borne by hand.

At the same time, **the human remains the source of decisions** wherever a commitment arises: signing the clarification protocol (client and vendor), signing the interpretation (the architect), QG-0 approval, the spot-check of tests, acceptance of the result. The AI agent executes; the human answers for the result. The client never talks to the agent directly: questions are aggregated and reworded by the architect.

Who Will Find RENAR Useful

RENAR is built for **contract-oriented development**: projects with an explicit contractual TZ and an identifiable client party answerable for that TZ. Typical contexts:

- **Custom development** — an independent vendor + a client with a signed TZ and acceptance criteria.
- **Regulated industries** (healthcare, finance, the public sector) — where a compliance audit is mandatory by regulation.
- **Enterprise consulting** — a third party implements against a corporate client's TZ with approval from several stakeholders.
- **Public-sector / government IT** — tender TZs, formal acceptance, multi-year contracts.
- **Long-lived products** — where the Product Owner plays the role of the Client's representative for internal feature TZs.

RENAR is **not applicable** to lean startup discovery, pure R&D without a defined scope, hackathon proofs-of-concept, and other contexts without an immutable TZ and an identifiable Stakeholder.

Routes by Role

Role	Where to start
PM / RTE	guide/05 — RENAR vs SAFe; then guide/09 §E3 — a practical example
Legal / Compliance	guide/09 §E3 → guide/06 → reference/07 — ISO 29148 mapping
Regulator / Auditor	reference/07 → reference/08 → standard/13 — conformance manifest
RE engineer / Architect	guide/00 Quickstart → standard/06 → standard/10

Where to Read Next

Document	Purpose
standard/ — 15 normative chapters	The complete normative description; required reading for the AI agent and the assessor
guide/00-quickstart	A 30-minute practical end-to-end example: TZ → ADAPT → SR → SPEC → TC
guide/01-walkthrough	An extended example on a full-scale scenario
guide/06-compliance	GDPR, FZ-152, AI Act mapping
reference/01-glossary	The canonical glossary + mapping to ISO 29148, BABOK, SAFe, NIST AI RMF
reference/02-schemas	Machine-readable artifact schemas (JSON Schema), validation rules
reference/03-ai-risk-register	14 AI risks per ISO/IEC 23894 + NIST AI RMF

00. Introduction

> ***Part of the RENAR Standard v1.0*** *· ← Table of contents*

> ***For newcomers.*** *This chapter is normative and dense (RFC-2119, closed lists, mandatory clauses). If you are a human meeting RENAR for the first time — start with the conceptual overview* `core/renar-core.md` *(≤ 10 min), then* `guide/00-quickstart` *(≈ 30 min), then come back here. If you are an AI agent — go straight to the normative chapters.*

0.1 Why this chapter

The client wrote in the TZ: "the user exports a report." The engineer read it as a PDF export, the specification named CSV, and the test ends up checking Excel. There is no ill intent — the requirement simply lives in five places at once (the client's contractual TZ, the engineering decomposition, the specifications, the test cases, the code), and at each seam the meaning shifts a little. When artifacts are written by a mix of humans and AI agents, there are more seams and the shift is faster. So the bottleneck of development is no longer code speed but **requirements correctness**. RENAR exists so that meaning does not leak at these seams: it sets explicit contracts between artifacts.

This chapter answers four questions: **what** RENAR is (§0.2), **why** it is needed (§0.3), **how** it relates to SENAR (§0.4), and **what the minimum** is below which conformance cannot be claimed — the Minimum Viable RENAR (MVR; §0.5). Closed lists and the language of the corpus — §0.6, §0.7.

This chapter introduces **no** new normative requirements. Each of the seven MVR statements is a reference to a §1–§13 chapter where it becomes a precise norm; here it is given as a coherent whole.

> ***Modal-verb convention.*** *Normative force is expressed with UPPERCASE RFC-2119 keywords:* `MUST` */* `SHALL` */* `REQUIRED` *— the mandatory level;* `SHOULD` */* `RECOMMENDED` *— the recommended level;* `MAY` */* `OPTIONAL` *— the permitted level;* `MUST NOT` */* `SHALL NOT` *— prohibition. Conformance follows RFC 2119 + ISO/IEC/IEEE 29148:2018 §5.2.1. This convention applies in all normative chapters. (The RU corpus uses lowercase modals as its canonical form under the RFC 8174 carve-out; the EN corpus uses UPPERCASE — see the EN Style Guide §2.)*

0.2 What RENAR is

RENAR (*Requirements Engineering & Normative Adaptive Regulation*) is a normative requirements-management standard for development with AI agents. The standard governs:

- The **data model** of requirement artifacts (BR / SR / TR), ADAPT, ACTZ, the eleven SPEC types, TC, and AT.
- The **lifecycle** of artifacts through a closed list of Quality Gates (QG-0 / QG-1 / QG-2 mandatory; QG-3 / QG-4 optional).
- The **substrate capabilities** V1–V6.
- **Conformance** — the RENAR-1..RENAR-5 levels, mandatory clauses, the manifest, and assessment procedures.

RENAR is a **specialization of SENAR** (§1.6) in the requirements-engineering domain. A RENAR-conformant implementation is **always** SENAR-compatible; the converse does not hold (§1.6.2).

RENAR is not tied to a specific substrate (VCS, document store, wiki): the normative chapters use substrate-independent language and govern the **capabilities** V1–V6 (§3.1). RENAR is a standard, **not** an implementation.

0.2.1 The AI agent as a regular implementer

RENAR artifacts are **routinely** created and maintained by an AI agent under an engineer's assignment. The human acts as reviewer and approver. This is a normative positioning, not a methodological recommendation.

Consequences:

1. **Completeness** — the completeness requirements for artifacts: the AI agent executes "code in natural language," and without completeness, provenance breaks down (§0.3).
2. **frontmatter density** — the dozens of fields are a consequence of the machine nature of the primary reader. In a multilingual UI, fields MAY be displayed human-readably (§4.13.3).
3. **Compensating mechanisms** — the ACTZ dual signature (§7.13), the Architect's signature on an ADAPT (§7.5), test-authorship isolation and the red history (§9.18), the engineer's spot-check (§9.14), adversarial review (§7.10.2), and the Quality Gates (§10.3) ensure that the human remains the source of decisions on contractual outcomes.

What §0.2.1 does **not** assert: RENAR does not require an AI agent for conformance — the standard is implementable by hand, but the process overhead makes that impractical (§1.4.1, indicator 5). The AI agent does not replace human approval — all normative signatures are performed by a human. The AI agent acts as responsible (R in RACI §5.6), not accountable (A).

0.3 Why RENAR exists

In development with AI agents, requirements live simultaneously in several artifacts created and maintained by a mix of human and AI-agent authors. Without normative contracts between them, **requirements drift** arises — a divergence between what is recorded, what is verified, and what is implemented.

RENAR closes **eight normative drift classes** (the full list with enforcement points — §4.11; the conceptual description — core/renar-core.md). All eight are **structural**: they arise from the very fact that an artifact is jointly owned by several authors, not from lapses in discipline. They can be closed only normatively — by fixing the contract of links, field authorship, and transition preconditions (§13.3, mandatory clauses).

The contract is machine-enforceable only on the condition that artifacts are complete (§0.2.1). An artifact with defaults is a contract with holes through which drift passes regardless of any hooks: a hook sees only what is recorded. The same applies to canonical names and closed lists (§4.2): a hook compares strings, it does not interpret synonyms.

0.4 Relationship to SENAR

RENAR governs **only** those aspects of requirements engineering that SENAR leaves to the domain standard's discretion: the artifact data model, the lifecycle of requirement artifacts, and the conformance

manifest for requirements engineering. RENAR does **not** duplicate and does **not** override SENAR constructs (the 5 values, 14 rules, common Quality Gates, 5 base roles).

The project conformance manifest (§13.4) MUST declare both `senar-version` and `renar-version`. A claim of RENAR conformance without a compatible SENAR version is non-conformant (§13.8).

If a RENAR normative statement turns out to be incompatible with a SENAR norm, that is a bug in the RENAR Standard. The resolution is through the formal change procedure of the SENAR Standard, with a corresponding alignment in RENAR (§13.9), not through a project-local override.

Aspect	SENAR (base)	RENAR (requirements-engineering specialization)
5 values + 14 rules + 5 roles	governs	inherits
Common Quality Gates	general framework	closed list of canonical QG-0..QG-4 (§10.3)
Artifact data model	— (out of scope)	BR / SR / TR / ADAPT / ACTZ / 11 SPEC types / TC / AT (§6–§9)
Requirement-artifact lifecycle	—	canonical state machines (§10)
Substrate capabilities	—	V1–V6 (§3)
Conformance manifest	generic SENAR	RENAR-CONFORMANCE.yaml + mandatory clauses (§13)

RENAR's original contribution (not inherited from SENAR): the requirements data model; ADAPT with two-way adaptation and the Architect's signature; ACTZ with the dual signature and the effective TZ as the acceptance baseline; substrate-independent V1–V6; the canonical lifecycle and QG for the requirements axis; pos/neg pairing and judge ≠ production as blocking clauses; the RENAR-1..RENAR-5 conformance levels.

0.5 Minimum Viable RENAR (MVR)

Minimum Viable RENAR is a closed list of seven normative statements, mandatory for any RENAR-conformant implementation regardless of the declared level (including `RENAR-1`). The MVR is equivalent to the mandatory clauses of §13.3.

#	Statement[1]	Normative source
MVR-1	**Source-of-Truth inversion**: the requirement-artifact hierarchy MUST be the source of truth about system behavior; code is a derived artifact. Reverse-engineering behavior from code into an SR without a bug-fix justification is prohibited.	§2.3, §13.3.1
MVR-2	**V1–V6 capabilities**: the project substrate MUST satisfy all six capabilities — immutable history (V1), atomic change unit (V2), diff & review (V3), branching / change-set (V4), cross-substrate version pin (V5), author + timestamp (V6).	§3.3, §13.3.2
MVR-3	**Reactive, stage-agnostic ADAPT (0..N per TZ)**: ADAPT is a reactive artifact, created if and only if converting a TZ → RENAR description **at any derivation stage** produces a	§7, §13.3.3

	gap between the client's language and the requirements language. A single TZ has **zero or more** ADAPTs; each is bound to its trigger (TZ import or a specific decomposition stage) through `trigger-stage`, and multiplicity is the regular case. When a gap is present, the ADAPT is REQUIRED to be in status `approved` with the Architect's signature; the client's decisions are recorded in signed ACTZ (§7.13), and the acceptance baseline is the effective TZ = the initial TZ + all signed ACTZ (§7.14). When no gap is present (the adversarial reviewer returned a "no findings" verdict), no ADAPT is created; BR/SR/SPEC reference the TZ directly through the mandatory `source.tz-section` + `source.adversarial-review-ref` → AR (§7.4.6). In either outcome the review result is issued as an AR. A delta-TZ follows the same rule.	
MVR-4	**Closed list of 11 SPEC types**: a SPEC type MUST belong to the closed list `SPEC-ARCH` / `API` / `DATA` / `INT` / `PROC` / `UI` / `AI` / `SEC` / `OPS` / `TEST` / `DOC`. A project MUST NOT create new types locally.	§8.3, §13.3.4
MVR-5	**TC pos/neg pairing**: every normative statement of a verifiable artifact that is covered by at least one TC MUST have a paired negative TC. The exception is when the statement itself describes a negative invariant.	§9.7, §13.3.5
MVR-6	**Closed list of Quality Gates**: an implementation MUST support QG-0 (Approval), QG-1 (Implementation), QG-2 (Verification) as `required`. QG-3 / QG-4 are `declared` or `absent`. Creating new gate types locally is prohibited.	§10.3, §13.3.6
MVR-7	**Conformance manifest**: a project MUST contain a manifest with both `renar-version` + `senar-version` + `level` (RENAR-1..RENAR-5) + confirmation of the §13.3 mandatory clauses. The manifest is immutable (V1).	§13.4

[1] Each row states a mandatory-level requirement (RFC-2119 `MUST` / `SHALL`) as interpreted by ISO/IEC/IEEE 29148:2018 §5.2.1.

An implementation satisfying all seven MVR conforms to at least the `RENAR-1` level (§13.2.1). An implementation that violates even one MVR **has no** RENAR conformance regardless of the declared level (§13.8).

0.6 Closed-list policy

The list of seven MVR is closed at v1.0. A project MUST NOT extend the MVR locally or shorten the list.

An implementation MAY **tighten** requirements beyond the MVR through the `declared-stricter` marker (§10.10.2, §13.4): require QG-3/QG-4 as `required`, require adversarial review of TC on all SPEC types, declare `RENAR-3+` as the minimum.

An implementation MUST NOT declare-weaker: issue a RENAR conformance claim without supporting one of MVR-1..MVR-7; declare ADAPT optional; allow single-TC coverage for a normative statement without the negative-invariant exception; omit `senar-version` or `level` in the manifest.

A change to the MVR is made only through the formal change procedure of the RENAR Standard (§13.9): research draft → public review → minor-version bump → migration guidance.

0.7 Corpus language

The RENAR normative corpus is bilingual: an RU edition (`standard/` , primary) and an EN edition (`standard/en/` , this file). Both are hybrids: **canonical identifiers** (latin and closed-list abbreviations) + connective prose in the edition's language. The policy is fixed by the EN Style Guide for the EN corpus and by reference/06 for the RU corpus, alongside §4.2. Artifacts and IDs, lifecycle statuses, VCS domain terms, and accepted technical terms stay latin in both editions; the editions differ only in the connective prose and in RFC-2119 polarity (EN UPPERCASE vs RU lowercase).

Substrate. The storage concept is, in the EN corpus, simply `substrate` — the canonical term used in prose, field names (`substrate-capabilities`), code/YAML, and file names alike. (The RU corpus renders the prose form as «носитель» and keeps `substrate` only in identifiers; see §4.2.)

Table of contents · **Next: 01. Scope →**

01. Scope

> ***Part of the RENAR Standard v1.0*** · *← Table of contents*

1.1 Where RENAR works and where it does not

Two teams write code with AI agents. The first builds a banking module against a signed TZ: there is a client, an acceptance, an audit — every requirement must be demonstrable. The second tests hypotheses at startup speed: a "requirement" exists today and is gone tomorrow after a pivot. RENAR is for the first. It is built where there is a contractual TZ and a party held accountable for it: contract development, regulated industries, consulting against someone else's TZ. In pure product discovery — "first build, then understand what was built" — RENAR is not merely redundant, it is structurally inapplicable: there is no immutable TZ, nothing from which to build an ADAPT. This chapter draws both boundaries — the **subject-matter** one (what the standard governs) and the **contextual** one (when it should be used at all) — through the closed lists of §1.2–§1.5.

The substantive norms of artifacts and the lifecycle are **not** set by this chapter — that is the domain of chapters 06–14; substrate-native implementation mechanisms are the domain of chapter 3 and `guide/03-tool-guide-*.md`.

1.2 What RENAR governs (closed list)

The closed list of RENAR's normative areas at version v1.0. Each area is covered in the indicated chapter:

#	Area	Chapter
1	Requirements hierarchy (BR / SR / TR)	06
2	ADAPT (two-way adaptation of the TZ): Forward interpretation + Backward findings + the Architect's signature; ACTZ with the dual signature	07
3	Closed list of 11 specification types (SPEC-ARCH / API / DATA / INT / PROC / UI / AI / SEC / OPS / TEST / DOC)	08
4	Test Cases as a full-fledged artifact (TC); pos/neg pairing; spec-specific TC obligations	09
5	Lifecycle states + Quality Gates (QG-0 / QG-1 / QG-2 mandatory; QG-3 / QG-4 optional)	10
6	substrate-independent versioning capabilities V1–V6	03
7	Maturity model (RENAR-1..RENAR-5)	11
8	Requirements-engineering metrics (RDLT, Hallucination Rate, DRA, ACR, etc.)	12
9	Conformance (mandatory clauses, manifest, self-assessment, third-party assessment)	13
10	Roles and responsibility for artifacts (specializations over SENAR §4)	05

All of the listed areas have normative content: mandatory requirements and a prohibition on project-local extensions of the corresponding closed lists.

1.2.1 What falls within the normative area

The RENAR normative area covers:

- **The artifact data model** — mandatory frontmatter fields, types, enum values, consistency invariants.
- **Lifecycle states + transitions** — state machines per artifact type; pre/post-conditions; the gate-id for each transition.
- **Identity and provenance** — immutable identifiers; substrate-native recording of authorship and time (V6); version pin between artifacts (V5).
- **Cross-artifact constraints** — `verifies[]`, `constrained-by[]`, `parent`, `delta-of`, `verified-by`, `implements-spec` — the normative semantics of links and the consistency rules.
- **AI provenance** — mandatory recording of the generator model, prompt-template, and token volume; the human-edits rules for approval.
- **Conformance procedures** — mandatory clauses, the manifest schema, the assessment cadence, the handling of conformance loss.

1.3 What RENAR explicitly does NOT govern (closed list)

The closed list of areas left **deliberately** outside the standard. Implementation in these areas is a free choice and does not affect conformance.

#	Area	What is out of scope
1	**The SENAR methodology as a whole**	The 5 values, 14 rules, common Quality Gates, agent instrumentation — governed by SENAR; RENAR neither duplicates nor overrides them.
2	**A specific substrate / VCS**	The choice of a specific version-control / document-database / wiki-platform is at the implementation's discretion; RENAR governs only the capabilities V1–V6 (§3).
3	**The implementation tech stack**	Programming languages, frameworks, databases, infrastructure components — out of scope; RENAR governs requirements, not the implementation.
4	**A specific UI / IDE / artifact editor**	Web interfaces, IDE extensions, CLI tools — out of scope; only the storage format of artifacts in the substrate is governed.
5	**Specific test runners**	pytest, jest, playwright, ragas, etc. — substrate-specific; RENAR governs only the mandatory recording of `automation.location` and `last-run` (V5+V6).
6	**Sales / contract business processes**	Pre-sale, formulation of the contractual price, the legal structure of contracts — out of scope; RENAR treats the TZ as an **input** and does not govern the process of its creation on the client side.
7	**Project management practices**	Agile ceremonies, sprint planning, kanban boards, story-point estimation — out of scope; RENAR governs the workflow of requirement artifacts, not management overhead.

8	**AI model selection and prompt engineering**	The choice of LLM provider, prompt-templates, fine-tuning strategies — out of scope; RENAR governs only the mandatory recording of `ai-provenance.generated-by` and adversarial review (§7.10.2).
9	**Specific substrate-native commands and hooks**	substrate-specific CLI commands (for example, specific VCS command names), the implementation of hooks (§10.11) — substrate-specific and belong in `guide/03-tool-guide-*.md`.
10	**Legal interpretation of artifact signatures**	Electronic signature, legal force, GDPR processing of personal data in the TZ — outside the standard's scope; governed by applicable law.

1.3.1 The substrate-independence principle

The RENAR Standard's normative chapters use substrate-independent terminology (§3.1). Substrate-dependent names (specific products, protocols, commands) do not appear in the normative text; they are present only in `guide/` (substrate-specific tools) and `reference/` (examples).

1.4 Scope of applicability (primary scope)

1.4.1 Contract-oriented development

RENAR's normative primary context is **contract-oriented development**: a project in which:

1. **A TZ exists** (an explicitly formulated requirement from the client side) which, once signed, becomes **immutable** (§7.4.1 ADAPT source).
2. **An identifiable client party exists** (a stakeholder with signing authority, §5.3.6) — able to confirm acceptance (§10.4.2) and to sign an ACTZ (§5.5).
3. **Reactive two-way client-side validation** — adversarial review of the TZ is mandatory; if findings are discovered or clarification is needed, the questions are put to the client as an ACTZ (§7.13), and the client's decisions are recorded in the ADAPT (§7.3, §7.4.1). If the conversion is unambiguous, validation reduces to a recorded "no findings" verdict (with no client interaction).
4. **A delta-TZ workflow** is possible — scope changes are formalized through a delta-ADAPT (§7.6), not through a hidden reinterpretation.
5. **An AI agent is available as the primary author** of RENAR artifacts (§0.2.1). Without an AI agent the standard remains applicable, but the process overhead of maintaining artifacts by hand — with full frontmatter, lifecycle transitions, and graph links — makes RENAR conformance impractical; the team either accepts that overhead or applies a `declared-stricter` limited scope (§1.7.2) to a critical subset of requirements.

1.4.2 Typical representatives of contract-oriented development

Context	Characteristics
Contract development against a TZ	Independent vendor + identifiable client; formal contract; acceptance criteria.
Regulated industries	A compliance audit is mandatory (healthcare, finance, public sector); traceability requirements → tests → code is required by regulation.

Enterprise consulting	A third party implements against the corporate client's TZ; approval by several stakeholders; an audit log.
Long-lived product with an explicit product owner	The Product Owner plays the role of the client's representative for internal feature TZs; an internal SLA + audit are mandatory.
Public-sector / government IT	Tender TZs; formal acceptance; multi-year contracts.

1.4.3 Spec-Driven Development (SDD) — the modern name

Contract-oriented development with AI acceleration is a form of **Spec-Driven Development** (§2.3.4). RENAR is the normative standard for AI-native SDD; it is not an alternative to SDD but its specialization in the requirements-management domain.

1.5 Where RENAR is inapplicable (negative scope)

RENAR is normatively inapplicable in contexts where the preconditions of §1.4.1 are structurally absent. A claim of RENAR conformance (§13.4) for projects in these contexts is non-conformant (§13.8).

1.5.1 Lean startup / pure discovery

A product team builds an MVP under market uncertainty; "requirements" are hypotheses validated against users, not immutable agreements. Out of scope: there is no immutable TZ (hypotheses are re-checked after a pivot); a client representative is structurally absent (the internal product manager = author + sole assessor — a violation of §5.5.3); the delta-TZ workflow does not apply (a pivot = a change of scope **as a whole**, not a delta). Lean startup teams MAY borrow **individual practices** of RENAR (AI provenance, adversarial review of TC) without claiming conformance — permitted as a source of ideas.

1.5.2 Pure R&D / research projects

A research project with no defined resulting scope (exploratory ML research, novel-algorithm prototyping without a client). Out of scope: there is no TZ as an immutable artifact; acceptance criteria (QG-4 §10.4.2) cannot be formalized — the "success" criterion of scientific research is not signed in advance; the ACTZ dual signature does not apply.

1.5.3 Exploratory hackathon / proof-of-concept

Time-boxed exploratory work with no mandatory client acceptance. The same reasons as §1.5.1 + an explicit waiver of formal acceptance.

1.5.4 Internal product without an external client

An internal team tool; the "client" coincides with the author; there is no independent stakeholder for the ACTZ dual signature. In the absence of an independent client representative, the ACTZ dual signature (§5.5) is structurally impossible — `client-signature.signed-by == vendor-signature.signed-by` violates §5.5.3. This is **the same underlying defect** as in §1.5.1: a structural absence of two-sidedness.

An implementation MAY apply a subset of RENAR practices **locally** (immutable identifiers, lifecycle states, V1–V6 for its own discipline), but it MUST NOT claim RENAR-N conformance. The manifest either does not

exist or explicitly declares "non-conformance."

Negative scenario: an attempt to declare RENAR-N for an internal product without an independent client representative is non-conformant (§13.8). If the internal product acquires an identifiable stakeholder (an internal Product Owner with acceptance authority), the scenario moves out of §1.5 into §1.4.2 "Long-lived product with an explicit product owner," and full conformance applies.

1.5.5 Negative scenarios — specifics

Scenario	Why it is non-conformant
A project with no written TZ; requirements are verbal discussions	Violation of §1.4.1 (1): there is no TZ as an immutable artifact; ADAPT has no source (§7.4.1).
A project where author == client (one person signs both sides)	Violation of §5.5.3 on two independent persons in the ACTZ signatures.
A project where scope is revised without a formal delta-TZ record	Violation of the §7.6 delta-ADAPT workflow; violation of the §13.3 mandatory clause §13.3.3 "Reactive ADAPT: adversarial review for every TZ, an ADAPT per the verdict."
A manifest that declares tech-stack-specific requirements (for example, "Python is mandatory")	Violation of §1.3 (3): the tech stack is outside the standard's scope; the manifest is non-conformant.

1.6 Relationship to SENAR

1.6.1 RENAR as a specialization of SENAR

SENAR is the **methodological base**: the 5 values of AI-native development, the 14 rules, agent instructions, common Quality Gates (§10.2.3), 5 base roles (§5.2).

RENAR is **a specialization of SENAR in the requirements-engineering domain**: it governs only those aspects that SENAR leaves to the domain standard's discretion (the artifact data model, the lifecycle, the conformance manifest). RENAR does not duplicate SENAR and does not override SENAR constructs.

1.6.2 Compatibility

A RENAR-conformant implementation is **always** SENAR-compatible. The converse does not necessarily hold: a SENAR-compatible project MAY **not** claim RENAR conformance if it operates outside the primary scope of §1.4.

An incompatibility of RENAR with SENAR at any normative point is a bug in the RENAR Standard; it is resolved through the formal change procedure of the SENAR Standard, with a corresponding alignment in RENAR.

1.6.3 RENAR is not an alternative to SENAR

An implementation MUST NOT claim "we follow RENAR instead of SENAR" — this is a violation of §1.6.1. RENAR is used on top of SENAR; the project's conformance manifest declares both the SENAR version and the RENAR version simultaneously (§13.4).

1.7 Closed-list policy (closed list)

1.7.1 What is closed at v1

The lists §1.2 (normative areas), §1.3 (exclusions), §1.4 (primary scope), §1.5 (negative scope) are closed. Project-local extensions and attempts to govern excluded areas through the manifest (§1.3 (3) tech stack, §1.3 (7) PM practices) are non-conformant. Adding new primary contexts or moving a scenario from §1.5 into §1.4 is done only through the formal change procedure of the standard.

1.7.2 Declared-stricter is permitted

An implementation MAY **tighten** scope relative to the normative minimum, declaring it explicitly in the conformance manifest (§13.4) with the `declared-stricter` marker (§10.10.2): apply RENAR only to a subset of requirements (security-critical SR); require additional artifact types (threat-models); prohibit RENAR without a Client representative. Declared-stricter is a permitted local policy; conformance to the normative level is preserved.

1.7.3 Declared-weaker is prohibited

An implementation MUST NOT declare-weaker relative to §1.2 / §1.3 / §1.4: claim RENAR conformance for a project in the §1.5 negative scope; apply RENAR without ADAPT (a violation of §13.3.3); govern excluded §1.3 areas through a project-local manifest.

1.7.4 The extension path

A change to §1.2 / §1.3 / §1.4 / §1.5 is done only through the formal change procedure of the standard (§13.9): a research draft with justification → public review → a minor- or major-version bump → migration guidance for existing conformant projects.

1.7.5 Master list of closed lists in RENAR

This paragraph is the single index of all closed lists in the RENAR Standard. Each listed list is non-extensible locally at the project level; changes are possible only through the formal change procedure of the standard (§13.9). The canonical source for each list is in the indicated section; the other mentions are cross-references.

#	Closed list	Canonical source	Cross-refs
1	Normative areas of the standard (10 items)	§1.2	§1.7.1
2	Exclusions from the normative area (10 items)	§1.3	§1.7.1
3	Primary scope: applicability contexts (5 indicators + 5 typical representatives)	§1.4	§1.7.1
4	Negative scope: inapplicability contexts (4 contexts + 4 negative scenarios)	§1.5	§1.7.1
5	RENAR roles (specializations over SENAR §4)	§5	§1.2 (10)

5bis	Requirement axis types (3 types: BR / SR / TR)	§6.2	§13.3
5ter	Decomposition levels (3: system / subsystem / module)	§6.3	§6.4, §4.9
6	ADAPT Backward findings categories (7 categories)	§7.4.4	§13.3.7
7	SPEC-* specification types (11 types: ARCH/API/DATA/INT/PROC/UI/AI/SEC/OPS/TEST/DOC)	§8.3	§4.4.1, §13.3.4
8	Normative TC principles (P1–P9)	§9.2	§13.3
9	TC types (`tc-type` : 6 values; `business` replaces the former `acceptance`)	§9.5	§4.5.2
10	Lifecycle states by artifact type (BR / SR / TR / SPEC / ADAPT / TC)	§10.5–§10.9	§4.7–§4.10
11	Quality Gates: mandatory {QG-0, QG-1, QG-2}, optional {QG-3, QG-4}	§10.3, §10.4	§10.10, §13.3.6
12	substrate-independent versioning capabilities V1–V6	§3.x	§1.2 (6)
13	Maturity levels RENAR-1..RENAR-5 (5 levels)	§11.3	§13.2, §13.9
14	REQ-specific metrics (11 metrics)	§12.3	§12.5
15	Drift classes (violations of the requirements infrastructure)	§4.11	§4.2
16	Prohibited / deprecated terms (non-canonical)	§4.14	§4.2

The extension of any closed list goes through the same formal procedure as a change to §1.2–§1.5 (§1.7.4, §13.9).

Chapter-level closed-list policies (§10.10, §12.3, §13.9) are specializations of this §1.7 for the corresponding lists and do **not** introduce independent procedures.

1.8 Cross-references

Source	Use
SENAR (full methodology)	RENAR's methodological base (§1.6.1); RENAR does not duplicate SENAR norms.
§5 Roles	Artifact ownership and roles — specializations over SENAR §4 (§1.2 (10), §1.6.1).
§2 Methodology positioning	Three foundational assertions (Source-of-Truth inversion, waterfall-form ≠ classical waterfall, substrate-independent versioning) — the justification of scope §1.4.
§7 ADAPT	ADAPT as the normative requirement §1.4.1 (3) for two-way client-side validation.
§10 Lifecycle and QG	Lifecycle + Quality Gates — the normative area §1.2 (5).
§3 Substrate versioning	The capabilities V1–V6 — the normative area §1.2 (6); the substrate-independence principle §1.3.1.
§13 Conformance	Mandatory clauses, the manifest, conformance loss — the normative area §1.2 (9); negative scenarios §1.5.5.

`core/renar-core.md`	A conceptual overview of the standard for a human reader (non-normative).
`guide/02-transition-guide.md`	Practical guidance for projects transitioning from a lean style to contract-oriented development (substrate-specific).

← Previous: 00. Introduction · **Table of contents** · **Next: 02. Methodology positioning →**

02. Positioning in the Typology of Methodologies

2.1 The three claims everything rests on

Before describing artifacts and processes, RENAR states the three ideas everything else stands on: **the Source of Truth is the requirements, not the code**; **the process has a layered form, but this is not classical waterfall**; **versioning is a mandatory substrate property, not a tie to a tool**. It sounds like generalities — but remove any one of them, and the remaining chapters fall apart. The requirements hierarchy (chapter 6) loses its Source-of-Truth meaning, ADAPT loses its role as the bridge between the contractual and engineering loops, the specifications lose their parallel axis, the test cases lose their binding to a requirement version, the lifecycle loses transition atomicity, and substrate versioning loses its normative footing.

This chapter is therefore the foundation; §2.3–§2.5 SHOULD be read in sequence. All three claims are **mandatory clauses**: each is a mandatory clause for any RENAR conformance claim (chapter 13).

2.2 The three fundamental claims

1. **Source-of-Truth inversion.** The Source of Truth about system behavior is the hierarchy of requirement artifacts. Code is a derived implementation artifact. This is Spec-Driven Development (SDD).
2. **Waterfall-shaped ≠ classical waterfall.** The RENAR process has a sequential, layered shape with gates. The standard explicitly distances itself from the four deadly sins of classical waterfall.
3. **Substrate-independent versioning.** Versioning is a mandatory property of the substrate implementing RENAR. The particular versioning tool is interchangeable. Six capabilities (V1–V6) set the normative requirements on the substrate; the details are in chapter 3.

The three claims are logically connected (see §2.6).

2.3 Claim 1 — Source of Truth about behavior: requirements, not code (Source of Truth)

2.3.1 Normative formulation

The Source of Truth about system behavior is the hierarchy of requirement artifacts: TZ → ADAPT → BR / SR / SPEC → TR → TC. Code is a derived artifact implementing this hierarchy. On a divergence between the code and a higher-level requirement, the requirement normatively wins.

2.3.2 Distribution of roles

Level	Who is the Source of Truth	Who is derived
TZ	Contract with the client	—
ADAPT	Two-way interpretation of the TZ	derived from the TZ
BR / SR / SPEC	The system's engineering standard	derived from ADAPT
TC	Contract of verifiable behavior	derived from SR / SPEC
TR (task)	The act of handing work to an implementer	derived from SR + SPEC
Code	Implementation	derived from everything above

2.3.3 Contract (mandatory consequences)

The standard requires the following behavior from any implementation claiming RENAR conformance:

1. **Prohibition of reverse-engineering behavior from code into an SR.** An AI agent or human creating a new SR or amending an existing SR uses ADAPT and the existing SRs as the source — but not the observable behavior of the implementation. Reverse-engineering is permissible only when creating a bug-fix task (see §2.3.4).
2. **Separation of code review and specification review.** Code review checks that a change conforms to the code. Specification review checks that the tests and the implementation conform to the requirements. These are two distinct gates, with distinct artifacts and distinct default reviewers.
3. **Drift detection as a substrate hook.** When a reference to an SR/SPEC absent from the requirements substrate is found in code, the atomic change unit in the implementation substrate is blocked (see chapter 10 §10.5, chapter 3 §3.5).
4. **Prohibition of silently adapting an SR to the code.** If the implementation does X while the SR requires Y, exactly one of two things is true: (a) the implementation is wrong — a bug-fix task is created to bring the code to the SR; (b) the SR is wrong — a delta-ADAPT is created with justification and signatures to change the SR. A third option ("we'll just update the SR to match the code") is prohibited.

2.3.4 Positioning in the industry typology

Claim 1 is a concrete realization of the **Spec-Driven Development (SDD)** paradigm — an industry term that emerged in 2024–2025 in response to the acceleration of development with AI agents. SDD recognizes that when AI agents can decompose formal specifications into code in minutes, **specification correctness** becomes the critical constraint, rather than code correctness.

Standard / methodology or framework	Corresponding provision	Relation to RENAR §2.3
ISO/IEC/IEEE 29148:2018 §6.4.5	Requirements management mandates traceability from requirements to implementation	RENAR §2.3 is a concrete realization of this mandate
BABOK Guide v3 §6.5	Verify requirements before they drive solution work	RENAR §2.3 normatively specifies verification as two distinct gates (code/spec)

ISO/IEC 5338:2023	AI-system lifecycle — requirements govern the generation of AI artifacts	RENAR §2.3, together with the reference/04 AI Style Guide and adversarial review (guide/07 §3.5), supports this mandate
Spec-Driven Development (industry, 2024-2025)	GitHub Spec Kit, Anthropic spec-first agents, Amazon Kiro, Tessl, BMAD-Method	RENAR is the formal standard within this paradigm

What is new here for RENAR. The Source-of-Truth inversion itself is the definition of the SDD paradigm, not a RENAR invention. RENAR's contribution is its **enforcement**: the four contractual consequences of §2.3.3 (prohibition of reverse-engineering into an SR, separation of code review and specification review, the drift hook, prohibition of silently adapting an SR to the code), which turn the paradigm from a principle into verifiable normative requirements.

2.3.5 Differentiation from SDD tools

Industry SDD tools and RENAR are in one paradigm, but in different layers:

Axis	Industry SDD tools (Spec Kit, Kiro, Tessl, BMAD)	RENAR
Nature	Reference implementation plus ready-made tooling with a prescribed process	Formal normative standard (capabilities, lifecycle, invariants)
Substrate	tied to a specific tool / platform	substrate-independent (V1–V6); VCS / document-oriented store / other — interchangeable
Contractual loop	Usually absent	ADAPT: two-way adaptation of the TZ + the Architect's signature; ACTZ with the dual signature (§7)
Conformance	Undefined	Conformance claim via a manifest + mandatory clauses (§13)
Verification	Tests as a practice	pos/neg pairing + judge ≠ production as blocking gates (§9)

RENAR does not compete with these tools: an industry SDD tool MAY be a **substrate-native RENAR implementation** without losing the portability of the conformance claim (§14.5.2).

2.4 Claim 2 — Waterfall-shaped, not classical waterfall

2.4.1 Normative formulation

The RENAR process has a sequential, layered shape: TZ → ADAPT → BR/SR/SPEC → TR → implementation → TC run → accepted. This is a waterfall-like shape. The standard explicitly distances itself from the four deadly sins of classical waterfall and MUST NOT be interpreted as classical waterfall in the sense of Royce 1970.

This claim is a **positioning clarification** (removing a false analogy with classical waterfall), not a claim of novelty: after the four distancings of §2.4.2 the shape coincides with the V-model and ATDD. RENAR's

novelty lies in ADAPT, V1–V6, and the translation of practices into mandatory clauses. The claim remains a mandatory clause (§2.7) — without it, a reviewer forces an unsuitable template onto RENAR.

2.4.2 The four distancings from classical waterfall

Classical waterfall (Royce 1970, industry practice 1970–1990)	RENAR
One big "requirements → design → code → tests" pass per quarter or year	**delta-TZ workflow**: each change is a mini-cycle taking days or hours. The same shape repeats hundreds of times with AI acceleration. See chapter 7 §7.6 (delta-ADAPT)
Tests at the end of the cycle, after implementation	**TC is a full-fledged verification artifact** (chapter 9). Paired pos/neg TCs are created together with the SR/SPEC, not after the code. This is closer to the V-model and ATDD than to waterfall
One-way "throw the spec over the wall"	**ADAPT is a two-way document** by construction. Forward (engineering interpretation) + Backward (questions to the client). Prohibition of one-way handoff
The spec is written once, then untouchable; reality drifts	**Continuous reconciliation** through substrate hooks. code ↔ spec drift is detected automatically and lands in a delta-ADAPT. The spec is alive

2.4.3 Applicability

RENAR is applicable in the following contexts:

- Contract-oriented development (the presence of a contract with the client, a signed TZ).
- Regulated industries: regulatory conformance, medicine, fintech, the public sector, PII processing.
- Enterprise consulting (a third party builds the product against someone else's TZ).
- Projects with a high cost of requirement changes late in the cycle.
- Projects requiring an audit trail for conformance reviews (chapter 13).

RENAR is **not applicable** in the following contexts:

- Pure product discovery without a contractual context (lean startup, product-MVP "build first, understand what we are building later").
- Pure R&D without defined requirements.
- Prototyping with a lifecycle shorter than the time to write an ADAPT.

Applicability is documented as part of the conformance procedure (chapter 13).

2.4.4 Positioning in the industry typology

Methodology	Relation to RENAR
Classical Waterfall (Royce 1970)	RENAR distances itself on the 4 points of §2.4.2
V-model	RENAR is closer to the V-model: paired TCs, tests at the start of each layer
Scrum / Kanban	Not in conflict, but RENAR is a different axis (artifact-based, not process-based). A Scrum sprint MAY contain a RENAR delta-ADAPT cycle

SAFe Solution Intent	ADAPT is a concrete realization of Solution Intent for contract-oriented development. See guide/05-safe-comparison.md
BABOK Requirements Analysis	RENAR §2.3+§2.4 is a formal realization of BABOK §3+§6 for the context of development with AI agents

2.5 Claim 3 — Substrate-independent versioning

2.5.1 Normative formulation

Versioning is a mandatory property of the substrate implementing RENAR. The standard sets six capabilities (V1–V6) that the substrate MUST provide. The particular versioning tool is interchangeable: a substrate satisfying V1–V6 implements RENAR regardless of whether it is a distributed VCS, a centralized VCS, a document-oriented DBMS with conflict resolution, or another mechanism.

2.5.2 The six mandatory capabilities (overview)

The full normative formulation of V1–V6 is in chapter 3. At the level of typological positioning:

#	Capability	What it provides
V1	Immutable history	Any past state of an artifact is recoverable
V2	Atomic change unit	A "change" is one transaction; all or nothing
V3	Diff & review	A human sees a change and approves or rejects it before integration
V4	Branching / change-set	Drafts are separable from the approved truth
V5	Cross-substrate version pin	The implementation substrate pins the version of the requirements substrate
V6	Author + timestamp	Every change has an author and a time

2.5.3 A substrate that does not satisfy V1–V6 does not implement RENAR

The impossibility of implementing RENAR on a substrate without V1–V6 is structural, not operational: claim 1 (Source-of-Truth inversion) physically does not work without versioning, because:

- It is impossible to say "the implementation was built against the requirements as of date X" — provenance is lost (violation of V1, V6).
- It is impossible to build a delta-ADAPT — there is no base version against which to measure the delta (violation of V1).
- It is impossible to verify a TC — the `verifies[].requirement-version` field loses meaning without a stable version identifier (violation of V5).
- The requirement transition `verified → accepted` is impossible — the gate has nothing to rely on (violation of V1, V5).
- An audit trail of "what we delivered to the client under contract 2025-Q3" is impossible (violation of V1, V6).

Therefore a substrate without V1–V6 (a flat file server with file renaming as the versioning mechanism; a document store without conflict resolution; other systems without immutable history) **does not implement RENAR** regardless of its other properties.

2.5.4 Substrate-independent normative language

RENAR's normative paragraphs use substrate-independent language: "atomic change unit," "version pin," "author and timestamp," "diff & review" — not the names of specific tool primitives. This ensures the standard applies to any future substrate satisfying V1–V6. Substrate-specific details are in Chapter 3 §3.4 (the normative mapping table of V1–V6 × substrates), guide/03-tool-guide-git.md (distributed VCS), guide/04-document-store-substrate.md (document-oriented store).

2.5.5 Positioning in the industry typology

Standard	Corresponding provision	Substrate neutrality
ISO/IEC/IEEE 29148:2018 §6.4.5	Configuration management of requirements	substrate-neutral; governs capabilities, not tools
BABOK Guide v3 §5.3	Maintain requirements (for traceability)	substrate-neutral
CMMI-DEV CM SG2	Track and Control Changes	substrate-neutral
SAFe Solution Intent	Versioned artifact	Does not govern the substrate
ISO/IEC 5338:2023	AI artifact provenance	substrate-neutral; provenance is a capability

RENAR follows the same neutrality and explicitly fixes V1–V6 as a contract, which these standards left implicit.

2.6 The logical connection of the three claims

The three claims are connected directionally:

```
flowchart TD
    C1["Claim 1 — Source-of-Truth inversion<br/>requirements > code"]
    C3["Claim 3 — substrate versioning V1–V6<br/>provenance, pin, history, atomic change"]
    C2["Claim 2 — waterfall-shaped, not classical<br/>delta-TZ, paired TC, ADAPT, reconciliation"]
    COD["Contract-oriented development<br/>is viable with AI acceleration"]
    C1 -->|requires| C3
    C3 -->|enables| C2
    C2 -->|returns| COD
```

Without claim 1: the Source of Truth is diffuse, the spec drifts, audit is impossible. **Without claim 3**: claim 1 is declarative and physically does not work. **Without an explicit claim 2**: a reviewer forces unsuitable

templates onto RENAR (agile sprint without a waterfall shape, or classical waterfall without the 4 distancings) and rejects the standard on a false analogy.

All three claims are mandatory clauses (chapter 13). At the v1 level, a RENAR conformance claim requires accepting all three claims; without any one of them it is untenable.

2.7 Consequences for the conformance procedure

The claims of §2.3, §2.4, §2.5 are **mandatory clauses** for any conformance claim at the RENAR levels (RENAR-1..RENAR-5; see ch. 11, ch. 13). A team cannot claim "we implement RENAR-1 without Source-of-Truth inversion" (a contradiction in terms) or "we implement RENAR on a substrate without V1–V6" (the substrate is non-conformant); it MAY choose **not to claim** RENAR conformance in a context of non-applicability (§2.4.3) — this is normatively permissible. The concrete self-assessment checklist is in ch. 13.

2.8 Relationship to other chapters of the standard

Chapter	Relation
06 Requirements	The BR/SR/TR hierarchy is a concretization of the Source-of-Truth chain from §2.3.2
07 ADAPT	Two-way adaptation is a concretization of claim 2 (distancing from "throwing the spec over the wall")
08 Specifications	SPEC-* as a parallel axis is a consequence of the Source-of-Truth inversion for structural description
09 Test cases	TC as a full-fledged verification artifact is a concretization of claim 2 (distancing from "tests at the end")
10 Lifecycle and QG	The gates enforce claim 1 (drift hooks) and claim 2 (continuous reconciliation)
03 Substrate versioning	The detailed norms of V1–V6 (§2.5 is an overview)
11 Maturity	The RENAR-1..RENAR-5 levels extend the claims with additional requirements
13 Conformance	Self-assessment against the three mandatory clauses

03. Substrate versioning — V1–V6

3.1 The six substrate capabilities

The Source-of-Truth inversion from chapter 2 — the requirement wins, not the code — rests on a single physical condition: the substrate MUST faithfully remember what changed and when. If a past state can be rewritten after the fact, the phrase "the implementation was accepted against the requirements of version X" loses its meaning, and with it the whole acceptance contract collapses. So the six capabilities V1–V6 are not an infrastructure detail but the foundation on which the RENAR idea stands at all; that is why we put them up front, before the chapters on artifacts.

This chapter gives a **detailed normative formulation** of each capability: preconditions, postconditions, the relation to the trace chain of the Source of Truth, and examples of mapping onto common substrates. The conceptual rationale is §2.5 (statement 3, "Positioning in the methodology typology").

The concrete versioning mechanism — distributed or centralized VCS, a document store with conflict resolution — is **interchangeable**. RENAR governs capabilities, not tools.

3.2 Rationale for mandatoriness

Let us examine, one by one, exactly what breaks without each capability. The relation here is structural, not operational: it is not about team discipline, but about the fact that without the capability the corresponding property of truth simply cannot be expressed.

No capability	What becomes impossible	Relation in RENAR
V1 (immutable history)	"The implementation was built against the requirements as of date X"	provenance, audit log, acceptance gate
V2 (atomic change unit)	Guaranteed consistency of changes	delta-ADAPT as an atomic change unit (see §7.6)
V3 (diff & review)	Approval of requirements and specifications	QG-0, QG-ADAPT-approve
V4 (branching / change-set)	A draft is separable from approved truth	transitions `draft` → `review` → `approved`
V5 (cross-substrate version pin)	Pinning a requirement version from the implementation	the `verifies[].requirement-version` field in TC (§9)
V6 (author + timestamp)	provenance of every change	signatures in ADAPT, `ai-provenance` in SR/SPEC

Therefore a substrate without V1–V6 — a flat file server with no history, a document store without conflict resolution, any mechanism without immutable change tracking — **does not implement RENAR** regardless of its other merits. This is a structural constraint, not a question of team discipline.

3.3 Normative definitions of V1–V6

Paragraphs §3.3.1–§3.3.6 are formulated **independently of any concrete substrate**. The names of specific products appear only in §3.4 (example) and §3.6 (language illustrations).

3.3.1 V1 — immutable history

Capability (immutable history): the substrate MUST ensure that, for any artifact, any past state is addressable and recoverable without loss.

Preconditions: the artifact is registered in the substrate as a versioned object.

Postconditions: for any point in time T in the artifact's history there exists a stable version identifier through which the state at moment T is fully recoverable.

Without V1 it is impossible to:

- Recover the artifact's state as of the contract-signing date.
- Compare the current state with a baseline.
- Build an audit log for conformance (chapter 13 §13.4).
- Set a baseline point for delta-ADAPT.

Concrete realizations on different substrates — §3.4.

3.3.2 V2 — atomic change unit

Capability (atomic change unit): the substrate MUST ensure that any change to an artifact (or to a consistent group of artifacts) is recorded as a single transaction: all or nothing. Intermediate inconsistent states are not observable from the outside.

Preconditions: the substrate has a notion of a transaction (atomic change).

Postconditions: after an atomic change, either all edits are visible to an observer or none are. An intermediate "inconsistent" state MUST NOT exist.

Without V2 it is impossible to:

- Update a BR and its related SR consistently in one transaction.
- Guarantee consistency between a requirement and its `linked-tasks[]` metadata.
- Carry out ADAPT approval as a single atomic action (the Architect's signature + the `answered → approved` transition in one transaction); ACTZ signing is a separate atomic action carrying the dual signature.
- Roll back a change without a "half-applied" state.

3.3.3 V3 — diff & review

Capability (diff & review): the substrate MUST ensure that a proposed (but not yet integrated) change is representable as a diff against a baseline state, and that a human or AI agent with review authority can approve or reject it **before** it is included in approved truth.

Preconditions: the proposed change exists separately from approved truth (see V4).

Postconditions: before approval the change exists but is not considered part of the Source of Truth. After approval it becomes part of the Source of Truth as an atomic change unit (V2).

Without V3 it is impossible to:

- Run the Quality Gates: QG-0 — approval of requirements and specifications (§10.3.1); QG-ADAPT-approve — ADAPT approval (§7.9).
- Apply the Architect's signature to the ADAPT (§7.5) and the dual signature to the ACTZ (§7.13).
- Review code and specification independently (see §2.3.3 (2)).
- Use adversarial review as a gate.

3.3.4 V4 — branching / change-set

Capability (branching / change-set): the substrate MUST separate **work in progress** (WIP) from **approved truth** (the Source of Truth) so that several independent changes can be developed in parallel without affecting the Source of Truth.

Preconditions: the artifact is registered in the substrate.

Postconditions: for any artifact at a given moment there MAY exist (a) exactly one approved version (the Source of Truth) and (b) zero or more drafts — each of which is either integrated through V3 or rejected.

Without V4 it is impossible to:

- Separate the `draft` lifecycle status from `approved` (chapter 10).
- Run several delta-ADAPTs in parallel.
- Hold backward findings in `asked-to-client` without blocking approved truth.
- Evolve SPEC-* experimentally without affecting requirements derived from the implementation.

3.3.5 V5 — cross-substrate version pin

Capability (cross-substrate version pin): substrate A that uses an artifact of substrate B MUST be able to pin a specific version of substrate B's artifact as a stable cross-substrate identifier. This identifier MUST unambiguously recover the state of the artifact in substrate B.

Preconditions: substrate A and substrate B satisfy V1 (each has a stable version identifier).

Postconditions: for every pinned reference in substrate A to an artifact of substrate B there exists a pair `(artifact-id, version-id)`, and through it the full state of substrate B's artifact at the moment of pinning is recovered.

Without V5 it is impossible to:

- Use the `verifies[].requirement-version` field in TC (chapter 9 §9.4).
- Tie the implementation substrate (code) to a specific version of the requirements substrate (SR/SPEC).
- Guarantee: "this implementation was accepted against the requirements of version X."
- Compute the TC freshness metric (a pinned version older than the current one — the TC is stale).

3.3.6 V6 — author + timestamp

Capability (author + timestamp): for every atomic change unit (V2) the substrate MUST register an identifiable author (a human or an AI agent with a unique id) and a timestamp with a precision no coarser than one second.

Preconditions: the substrate has an author identification system.

Postconditions: for any atomic change unit the query "who? when?" yields an unambiguous answer.

Without V6 it is impossible to:

- Apply the ACTZ dual signature (a signature = author + timestamp in the substrate's native form); an ADAPT is signed by the Architect alone.
- Record `ai-provenance` in artifact frontmatter.
- Build an audit log for conformance.
- Compute the adversarially-found metric (the distribution of backward findings across authors).

3.4 Mapping V1–V6 onto concrete substrates (example)

> ***Informative.*** *The table shows how V1–V6 are usually realized on common substrates. It is* ***not*** *part of the normative contract. Any substrate that satisfies V1–V6 implements chapter 3 — whether or not it appears in the table.*

Capability	Git	Mercurial	SVN	Perforce	Document store (example)
V1 — immutable history	commits, hash-chain	changesets, hash-chain	revisions, sequential numbering	changelists	revision tree per doc (`_rev`)
V2 — atomic change unit	commit	commit	atomic revision	changelist submit	document update (single _rev advance)
V3 — diff & review	merge request / pull request	hg phabricator, mq	svn diff + commit gate	swarm review	API workflow + Hub UI approval
V4 — branching / change-set	branches	named branches, bookmarks	branches (copy semantic)	branches / streams	conflict branches / WIP docs / draft status
V5 — cross-substrate version pin	submodule SHA	subrepo changeset	externals (peg rev)	branches / streams reference	`_rev` reference + `created_by_order`
V6 — author + timestamp	commit metadata	commit metadata	revision properties	changelist metadata	doc fields (`author` , `updated_at`)

Practical workflows for each substrate:

- guide/03 — git
- guide/04 — document store

3.5 A substrate that does not satisfy V1–V6

The following configurations **do not implement RENAR**, because they violate one or more capabilities:

Configuration	Violation
Flat file server; "version" = renaming the file	V1 (no stable history; renaming = loss of provenance)
Document store without conflict resolution	V2 (an inconsistent state is possible)

Wiki without revision history	V1, V6
Wiki with revision history but no approval procedure	V3
VCS using `mtime` instead of an immutable identifier	V1, V5
Substrate that edits historical revisions in place	V1 (history is mutable — an audit log is impossible)

A team **adopting** RENAR on a substrate from this list **MUST** either migrate to a suitable substrate or build a **compensating layer** that provides V1–V6 on top of the underlying storage. In both cases the conformance manifest MUST explicitly document how V1–V6 are realized.

3.6 Substrate-independent language

RENAR normative paragraphs use **substrate-independent** wording: "atomic change unit" (V2), "version pin" (V5), the `approved` state (after V3), "draft" (V4). Terms **specific to a substrate** (tool names and their primitives) are permitted only:

- in illustrative tables with an explicit marker (as in §3.4);
- in cross-references to a substrate-specific guide/;
- in appendices to normative chapters.

3.6.1 Examples: normative form and a git illustration

Normative (substrate-independent) wording	Git illustration (not the norm)
"A requirement change is recorded as an atomic change unit with author and time (V2, V6)"	"The change is recorded as a commit"
"A change passes diff & review before integration (V3)"	"The change passes merge-request review before merge into main"
"The implementation substrate pins a specific version of the requirements substrate (V5)"	"`.src` pins the `.req` submodule SHA"
"The ACTZ dual signature — two independent author+timestamp events (V6)"	"The dual signature — two approvals in the merge-request UI"
"Editing an already-approved requirement outside an atomic change unit with review is a violation"	"A force-push to an approved branch is blocked by hooks"

3.6.2 The substrate as a conformance parameter

Each team fixes its **substrate choice** in the conformance manifest with an explicit mapping V1–V6 → substrate-native primitives. When the substrate changes (for example, from git to a document store) the manifest is updated and a regression check of V1–V6 is run (§3.8).

3.7 Conformance manifest

The canonical schema is §13.4, the file `RENAR-CONFORMANCE.yaml` (YAML 1.2) at the root of the requirements substrate. Chapter 3 governs the substrate-related part: the declaration of the chosen substrate and the V1–V6 mapping (the `substrate-capabilities` field, §13.4.2).

```yaml
# Fragment of RENAR-CONFORMANCE.yaml — the substrate-specific part
# (full schema — §13.4.2)
substrate-capabilities:
  v1-immutable-history:  declared    # the substrate primitive and how it is
checked — via substrate-id
  v2-atomic-change-unit: declared
  v3-diff-review:        declared
  v4-branching:          declared
  v5-version-pin:        declared
  v6-author-timestamp:   declared
  substrate-id: "<substrate-native pointer to guide/03..06>"
```

Confirmation of the mandatory clauses (§2.3 Source-of-Truth inversion, §7 ADAPT, §8 closed list of 11 SPEC types, §9 pos/neg, and others) goes in the `mandatory-clauses-confirmed` field (§13.3–§13.4).

The manifest is **REQUIRED** for any conformance claim at level RENAR-1 and above (chapter 13).

3.8 Substrate migration

When changing the substrate, the team **MUST** perform the steps in order:

1. **Pre-migration audit:** the target substrate satisfies V1–V6; otherwise migration is prohibited until a compensating layer exists.
2. **Manifest draft:** an updated `RENAR-CONFORMANCE.yaml` with the new mapping.
3. **Isomorphism check:** all artifacts (BR, SR, SPEC, ADAPT, TC) are transferable without loss of fields, ids, or provenance. All ids are **immutable** — renaming during migration is prohibited.
4. **Atomic switchover:** the migration is an atomic change unit at the process level; a single-moment transfer of all artifacts. **Using two substrates as the Source of Truth in parallel is prohibited** (see §2.3.3 (1)).
5. **Post-migration check:** a regression check of V1–V6 on real artifacts.
6. **Manifest registration:** the updated `RENAR-CONFORMANCE.yaml` is registered in the new substrate as an atomic change unit (V2).

After switchover the old substrate is an archive (a read-only snapshot) or is decommissioned. Working on two substrates as the Source of Truth in parallel is prohibited.

3.9 Relation to other chapters

Chapter	Relation
02 Positioning in the typology §2.5	Conceptual rationale for V1–V6

06 Requirements hierarchy	frontmatter relies on V1 (immutable id), V5 (version pin)
07 ADAPT	ADAPT approval by the Architect's signature — V3 + V6; the ACTZ dual signature — V3 + V6; delta-ADAPT — V1 + V2 + V4
08 Specifications	`constrained-by[]`, `depends-on[]`, `referenced-by[]` — through V5
09 Test cases	`verifies[].requirement-version` — a direct application of V5
10 Lifecycle and QG	Status transitions — V2 + V3 + V4
11 Maturity model	RENAR-3+: V5 required; RENAR-4+: V6 + ai-provenance
13 Conformance	`RENAR-CONFORMANCE.yaml` — a mandatory artifact
guide/03	Practice on git
guide/04	Practice on a document store

04. Terms and Definitions

> ***Part of the RENAR Standard v1.0*** · *← Table of contents*

4.1 Why a single vocabulary of terms

The same word often means different things to two teams: for one a "spec" is an SR, for another a screen mock-up, for a third a whole API contract. As long as terms float, traceability collapses and any conversation about conformance stalls. This chapter removes the ambiguity: a reference of phrasings read when aligning artifacts, checking the substrate, and preparing a conformance assessment. It fixes one name per concept and so quenches terminological drift between teams and tools. After the table of contents, go to §4.3–§4.5 for artifact types, §4.6–§4.7 for statuses and gates, §4.8–§4.10 for substrate and provenance, §4.11/§4.14 for drift classes and forbidden terms; the index of closed lists — §1.7.5.

The chapter normalizes the **canonical terminology of RENAR**: one definition per concept; a single source of truth for the other chapters, implementation substrates, and conformance-checking tools. Terminological drift (§4.11) is a distinct class of conformance violations (§13.3.1 indirectly).

The chapter does **not** duplicate reference/01-glossary.md: this chapter contains the **canonical normative** short-form definitions; `reference/01` provides expanded explanations with anti-patterns, history, and industry context (informational material).

4.2 The "canonical only" principle

RENAR picks **one canonical term** per concept. Inside the substrate (frontmatter, IDs, normative body paragraphs, scripts, CI hooks) **only the canonical term** is used. The mapping to related standards (§4.13) is for documentation, migration, and integration with external systems; inside the substrate, replacing the canonical term with an equivalent from the mapping table **does not happen**.

When a non-canonical term (per §4.14) is detected in a normative artifact, the substrate-native hook (§10.11.1) MUST raise a warning on the change-set; for RENAR-4+ (§11.7) — a blocking error.

Multilingual projects MAY display the canonical terminology in the UI in the client's language (§4.13.3); this is a **UI translation**, not a canonical replacement.

4.3 Requirement artifacts

4.3.1 TZ — source requirements artifact

TZ (`TZ-YYYY-NNN`) is an **immutable** (§7.4.2) contractual artifact recording the obligations between the client and the engineering team. Once registered in the substrate it is not edited. Changes go through a delta-TZ as a new immutable artifact (§7.6).

4.3.2 ADAPT — bridge artifact

ADAPT (`ADAPT-NNN`) is a **reactive** bridge artifact (§7.4.1) between the TZ and the requirements hierarchy. It contains Forward (the engineering interpretation) + Backward (questions to the client) sections. It is created if and only if the adversarial review returned a "findings present" verdict at some derivation stage; in that case the ADAPT MUST be in status `approved` with the Architect's signature (§7.5); the client does not sign an ADAPT — the client's decisions live in ACTZ (§4.3.7). A TZ has **zero or more** root ADAPTs (cardinality 0..N, MVR-3, §13.3.3); on a "no findings" verdict no ADAPT is created and artifacts reference the TZ directly (§4.10.4 AR). Lifecycle: §4.6.4.

4.3.3 BR — Business Requirement

BR (`BR-NN`) is a business-level artifact. It describes an observable business effect (`business-outcome`), not the way of achieving it. It decomposes into SR. Frontmatter — §6.5.2. Lifecycle: §4.6.1.

4.3.4 SR — System Requirement

SR (`SR-NN`) is a system-level artifact. It describes the mandatory behavior of the system within a single business effect. It has a parent BR (`parent: BR-N`) or a parent SR (when `level: subsystem` or `level: module` — §6.7). Frontmatter — §6.6.2. Lifecycle: §4.6.1.

4.3.5 TR — Task Requirement

TR (`TR-NN`) is an implementer task-level artifact. It describes a practically executable piece of work with goal + AC. It has an implementation chain `implements: SR-N` (or BR for trivial tasks). Frontmatter — §6.7.2. Lifecycle: §4.6.2.

4.3.6 Hierarchy

The requirements hierarchy:

```
TZ → ADAPT → BR → SR → TR → implementation
                    │
                    └── SPEC-* (parallel axis, §4.4)
```

An SR is constrained by SPECs through `constrained-by[]` (§8.6.1) — this is a **link**, not a parent edge; the `implements-spec[]` edge belongs to TR (§8.6.2).

4.3.7 ACTZ — the TZ clarification protocol

ACTZ (`ACTZ-NNN` , Agreed Clarification of TZ) is an artifact of the **contractual loop**: a batch of questions, proposals, and decisions put to the client and signed by **both parties** (§7.13). Its contractual name is "**TZ Clarification Protocol No. N**". It carries contractual weight; once signed it is immutable. Cardinality: 0..N per TZ, 1..N per ADAPT. Lifecycle: `draft → sent → signed → superseded` .

The boundary with ADAPT is drawn **by audience**: what is put to the client and approved is an obligation (ACTZ); what is not put to the client is an interpretation (ADAPT).

4.3.8 Effective TZ

The **effective TZ** is the acceptance baseline (§7.14): the initial TZ (including its annexes) **plus all signed ACTZ**. On a discrepancy, the later signed document prevails. The acceptance tests AT (§4.5.4) are derived from the effective TZ. ADAPT is not part of the acceptance baseline.

4.4 SPEC artifacts

SPEC is an artifact of the structural description of the system as a parallel axis to the requirements (§8.2). It is not a parent edge with SR; it is linked through `constrained-by[]` / `implements-spec[]` (§8.6). Lifecycle: §4.6.3.

4.4.1 Closed list of the 11 SPEC types

Closed list (§8.3):

Type	Purpose
`SPEC-ARCH`	System architecture (contexts, containers, components, deployment view, quality attributes)
`SPEC-API`	API contracts (REST / GraphQL / gRPC / async events)
`SPEC-DATA`	Data model (schema, ERD, migrations, retention, PII classification)
`SPEC-INT`	Integration (interaction between subsystems and external systems)
`SPEC-PROC`	Process / workflow (business processes, state machines, saga)
`SPEC-UI`	UI / UX (screens, navigation, accessibility, baselines)
`SPEC-AI`	AI / ML (model cards, RAG, prompt engineering, eval strategy)
`SPEC-SEC`	Security (authn / authz, threat model, secrets management)
`SPEC-OPS`	Operations (deployment, observability, SLO/SLA, runbook) — **how the system lives**
`SPEC-TEST`	Test benches and data (topology, emulators, datasets on both sides of the integrations, reconciliation rules) — **how conformance is proven** (§8.3.2)
`SPEC-DOC`	Delivered documentation (composition, structure, checkable requirements) — **what enters the delivery**

A project MUST NOT create new SPEC types locally (§13.3.4).

4.5 Testing artifacts

4.5.1 TC — Test Case

TC (`TC-NN`) is an artifact of a verifiable criterion. It covers the normative assertions of BR / SR / SPEC (through `verifies[]` with a version pin, §9.3). Lifecycle: §4.6.5.

4.5.2 Closed list of TC types (`tc-type`)

Closed list (§9.5):

`tc-type`	Purpose
`business`	E2E tests of the business goal for BR (§9.5); runner-family: E2E + AI validator. The former name `acceptance` is withdrawn: there are two acceptances, and one name for two different subjects caused confusion
`ux`	UX tests with a VLM judge (§9.6.1) for SPEC-UI
`system`	General-purpose system tests for SR / SPEC-PROC / SPEC-ARCH (§9.5)
`contract`	Contract tests (§9.6.3) for SPEC-API / SPEC-INT / SPEC-DATA
`eval`	Eval tests for SPEC-AI with an LLM judge (§9.6.2); the judge model MUST differ from the implementation model
`security`	Security tests (§9.6.4) for SPEC-SEC by STRIDE categories

4.5.3 Pos / neg pairing

Normative requirement (§9.7): every normative assertion is covered by a **pair** of TC (positive scenario + negative scenario). Single-TC coverage is permitted only for invariant assertions (security STRIDE).

4.5.4 AT — Acceptance Test (the acceptance test of the contractual loop)

AT (`AT-NN`) is an acceptance test derived by an **isolated agent exclusively from the effective TZ** (§4.3.8), with no access to ADAPT / BR / SR / SPEC / TC / code (§9.19). It checks conformance to the **contract**, not to the interpretation. It is regenerated before every acceptance trial from the current revision of the effective TZ. The divergence "AT red while TC are green" diagnoses an **interpretation error** — a defect class that TC cannot catch in principle.

4.6 Lifecycle statuses

4.6.1 BR / SR

Closed list (§10.5): `draft → approved → verified → accepted → deprecated` . `accepted` is a terminal non-degradable status (§10.4.2, optional); `deprecated` is terminal.

4.6.2 TR

Closed list (§10.6): `draft → approved → done`; `obsolete` is the alternative terminal status.

4.6.3 SPEC

Closed list (§10.7): `draft → review → approved → verified`; `obsolete` is terminal.

4.6.4 ADAPT

Closed list (§10.8): `draft → review → asked → answered → approved → frozen`, plus the terminal `superseded` on supersession (§10.8.5). `frozen` is a terminal immutable status; changes are made only through a delta-ADAPT, errata, or supersession.

4.6.5 TC

Closed list (§10.9): `draft → ready → passing / failing → obsolete`. `passing ↔ failing` are runner-managed transitions (§10.9.3), not gate-passages.

4.6.6 Backward findings (sub-state in ADAPT)

Closed list (§7.4.5): `open → asked-to-client → answered → resolved → frozen`; `revised` is a return to `asked-to-client`.

4.6.7 Lifecycle closed lists are not locally extensible at the project level

Any status outside the closed list for the corresponding artifact type is a conformance violation (§10.10.2).

4.7 Quality Gates

The canonical list from §10.3 + §10.4. A project MUST NOT create new gate types locally (§10.10.2, §13.3.6).

Gate	Purpose	Conformance status
`QG-0` — Approval (§10.3.1)	Approval of an artifact for development / implementation	**Required**
`QG-1` — Implementation (§10.3.2)	Implementation valid (TC `draft → ready` only)	**Required**
`QG-2` — Verification (§10.3.3)	Promote an artifact to `verified`	**Required**
`QG-3` — Architecture (§10.4.1)	Approval of ADAPT (the Architect's signature) / SPEC-ARCH	Optional (`declared` or `absent`)
`QG-4` — Acceptance (§10.4.2)	Acceptance of a BR into `accepted`	Optional

Runner-managed transitions (`ready → passing` , `passing → failing` , and others) are **not** Quality Gates (§10.9.3).

4.8 Substrate terms

4.8.1 Substrate

Substrate (in fields and code — `substrate`) is the system for storing and versioning RENAR artifacts. RENAR is substrate-independent: it normalizes capabilities (§4.8.2), not tools.

4.8.2 Capabilities V1–V6

The closed list from §3.3. All six are absolutely mandatory for conformance (§13.3.2):

Capability	Semantics
`V1` — Immutable history	Any past state of an artifact is recoverable
`V2` — Atomic change unit	Changes are committed as "all or nothing"
`V3` — Diff & review	A proposed change is representable as a diff against a baseline state and passes approval before integration into the source of truth
`V4` — Branching / change-set	Drafts are separable from approved truth; parallel changes are independent
`V5` — Cross-substrate version pin	A link between substrates pins a specific version of an artifact
`V6` — Author and timestamp	Every atomic change unit has an identifiable author and a timestamp of ≥ second-level precision

4.8.3 Atomic change unit

A substrate change (V2) satisfying the "all or nothing" property — a substrate-native transaction; intermediate inconsistent states are not observable from the outside. The concrete implementation (an atomic write in a distributed VCS, a transaction in a document store, another mechanism) is substrate-specific; the description of the forms is deferred to `guide/` .

4.8.4 Version pin

A substrate-native mechanism (V5) that pins a specific version of an artifact in one substrate from another through the pair `(artifact-id, version-id)` .

4.8.5 Audit trail

A substrate-native append-only collection of gate-passage and transition events (§10.13). Each event contains a timestamp, artifact-id, artifact-version, from-status, to-status, gate-id, actor, evidence-refs.

Deletion is not permitted (V1 retention, §10.13.3).

4.9 System hierarchy

Closed list of levels (§6.7, §6.8):

Level	Purpose
`system`	The entire product as a whole; the top level; rarely used (cross-subsystem tasks)
`subsystem`	A subsystem (for example, a separate service, a frontend application)
`module`	A module within a subsystem

The `level` field is recorded in the artifact frontmatter (BR / SR / TR). The hierarchy MAY be extended downward through subsystem → module evolution (§6.9.1) or back upward (§6.9.3).

4.10 Provenance terms

4.10.1 ai-provenance — canonical schema

A frontmatter block (§11.7.1, mandatory at RENAR-4) recording the provenance of an AI-generated artifact. This section is the **single canonical source** of the schema; the chapter-level YAML examples in §6.5.2, §6.6.2, §8.5, §9.3 refer here and do **not** define independent fields.

Field	Mandatory?	Semantics
`ai-provenance.generated-by`	mandatory	Model identifier (`<vendor>-<model>-<version>@<date>`)
`ai-provenance.generated-at`	mandatory	UTC timestamp of generation (ISO-8601)
`ai-provenance.prompt-template`	mandatory	Substrate-native pointer to a prompt template (`<path>@<version>`)
`ai-provenance.context-tokens`	mandatory	Input context size (integer)
`ai-provenance.output-tokens`	mandatory	Output size (integer); source for the metrics in §12.3
`ai-provenance.human-edits`	mandatory	Boolean — whether manual edits were made after generation (informational field; see §4.10.1.1)
`ai-provenance.generation-time-ms`	optional	Generation latency in milliseconds; RECOMMENDED at RENAR-5 for cost/latency budget monitoring (§11.8.1)
`ai-provenance.cost-budget`	optional at RENAR-4, mandatory at RENAR-	Planned generation cost budget

	5	
`ai-provenance.cost-actual`	optional at RENAR-4, mandatory at RENAR-5	Actual cost; source for §12.3.9 Cost per Approved Requirement

Adding new fields to the schema is done only through the formal change procedure of the standard (§13.9). Locally defined ai-provenance.* fields by an implementing project are a `declared-stricter` extension (§10.10.2) and do not violate conformance, but are not considered canonical.

4.10.1.1 Semantics of `human-edits`

`human-edits` is an **informational** field for traceability and observability, not a gating flag. The value `human-edits: true` means the artifact was edited manually after the initial AI generation; it does **not** trigger auto-rejection. The normative rule P3 (§9.2) — "the engineer does not write TC by hand" — normalizes **provenance**, not subsequent edits. A substrate implementation MAY (`declared-stricter`) additionally require a review for artifacts with `human-edits: true` ; this is a local tightening, not part of the base RENAR-N conformance.

4.10.2 Source citation

An inline pointer in the artifact body (mandatory at RENAR-4, §11.7.1) to the source of a specific normative assertion. The format is substrate-specific; typical patterns: `[TZ-XXX §Y line Z]` , `[ADAPT-NNN §A.B]` , a `derived` marker with a pointer to the parent artifact.

4.10.3 Traceability chain

The chain of artifacts from the TZ to a TC run, recording the provenance of each assertion:

```
TZ → ADAPT → BR → SR → TR / SPEC → TC.last-run
```

Each link is connected through canonical frontmatter fields (§4.12). The trace chain is the source of truth for the conformance review (§12.5.3).

4.10.4 AR — adversarial-review record

AR (`AR-NNN` , Adversarial-Review Record) is an **evidence record**, immutable once issued, that captures the fact and the outcome of the mandatory adversarial review of a TZ (§7.4.6). Mandatory fields: `tz-ref` , `trigger-stage` , `reviewer` (a model different from the primary agent's), `verdict` (`findings-present` or `no-findings`), `produces-adapt[]` (when `findings-present`), `status` , and a V6 signature. Lifecycle: `draft → issued → superseded` .

AR is **not** a requirements artifact and **not** a node of the graph: it is not decomposed, not verified by test cases, and belongs to no closed list of the standard (§1.7.5). Its purpose is to give the adversarial-review verdict an identity and make it auditable in both outcomes: on `findings-present` the AR points to the ADAPT it produced; on `no-findings` it serves as the evidence referenced by `source.adversarial-review-ref` (§4.12).

4.11 Drift classes (closed list)

A closed list of eight classes of violations of the requirements infrastructure. Changing the list is a formal change procedure of the standard (§13.9.3). The substrate-native hook (§10.11.1) MUST detect each class at the corresponding enforcement point.

#	Class	What is violated	Enforcement point
4.11.1	Schema drift	An artifact's frontmatter does not conform to the mandatory schema ch. 06/08/09	Substrate hook on the change-set; RENAR-3+ — blocking (§11.6.1)
4.11.2	Lifecycle drift	An artifact is outside the closed list of statuses (§4.6) or has gone through a forbidden transition (§10.12)	Substrate hook on the promote-transition
4.11.3	Source-of-truth drift	Implementation code / a derived artifact diverges from the SR / SPEC it references through `verifies[].version` (V5)	Reconciliation hook RENAR-4+; registered as a backward finding in a delta-ADAPT
4.11.4	Implementation drift	The implementation substrate has stopped referencing the current `version` of the requirements substrate (the V5 pin is stale)	Auto-invalidate `verified` (§10.5.4)
4.11.5	Terminological drift	Use of a non-canonical term (§4.14) in a normative artifact	Substrate hook on the change-set; RENAR-4+ — blocking
4.11.6	Order / provenance drift	An artifact references a source in a lower status than the §10.3.1 reference-validation requires	Substrate hook (§10.11.1) blocks the change-set
4.11.7	TC ↔ requirement provenance drift	A TC has lost its `verifies[]` reference or `last-run.requirement-version` does not match the current `version`	Runner-managed: the TC is moved to `failing` (§10.9.3) until a re-run
4.11.8	Test-fitting drift	A TC's Pass / Fail criteria are changed simultaneously with the implementation code so that a failing TC becomes passing without addressing the root cause (§9.13)	Substrate hook via the `[test-spec-change]` marker; a single person cannot approve both change-sets (§10.11.3)

4.12 Connection field terms (frontmatter)

Canonical names of the fields recording links between artifacts:

Field	Source artifact	Semantics
`parent`	BR / SR	A single parent in the hierarchy (BR-NN or SR-NN)
`children[]`	BR / SR	Auto-derived reverse edge (§6.x)
`implements`	TR	Implementation chain (`SR-N` or `BR-N`)

`implements-spec[]`	TR	Implementation of SPEC-* specifications (§8.6.2)
`constrained-by[]`	SR	Constraints from SPEC-* (§8.6.1)
`depends-on[]`	SPEC	Dependency graph between SPECs (§8.6.3)
`verifies[]`	TC	Closed list of verifiable artifacts with `version` (§9.3)
`verified-by[]`	BR / SR / SPEC	Auto-derived reverse edge from `verifies[]`
`source.adapt`	BR / SR / SPEC	The ADAPT from which the artifact was derived
`replaces` / `replaced-by`	Any	Replacement on deprecation (§10.5.3)
`supersedes`	A new version of an artifact	Which artifact is being replaced (in lieu of "reviving" an obsolete one)
`last-run`	TC	Result of the last runner run (§9.12); bot-managed only

The full schema of each artifact — in reference/02-schemas.md.

4.13 Mapping to related standards

4.13.1 Requirement artifacts

RENAR canonical	SENAR (RU)	ISO/IEC 29148	BABOK v3	SAFe
`BR` (Business Requirement)	БТ (Бизнес-требование)	Business Requirement	Business Need	Portfolio Epic / Strategic Theme
`SR` (System Requirement)	СТ (Системное требование)	System Requirement / Software Requirement	Solution Requirement (Functional)	Feature
`TR` (Task Requirement)	ТЗ (Требование к задаче)	(no direct class; detailing of a system / system-element requirement)	Solution Requirement (detailed)	Story
`ADAPT`	(RENAR-extension)	(n/a — formalised bridge artefact)	Stakeholder Requirement workshop output	(n/a)
`TC` (Test Case)	ТК	Test Case (verifiable item)	Verification artefact	Story acceptance test
`SPEC-*` (11 types)	(RENAR-extension)	Design Description (subset)	Solution Component (subset)	Enabler tech spec (subset)

4.13.2 Lifecycle statuses

RENAR canonical	ISO/IEC 29148	CMMI
`draft`	proposed	identified
`approved`	agreed-to / baselined	committed
`verified`	verified	validated
`accepted`	accepted	accepted
`deprecated` / `obsolete`	retired / superseded	obsolete / superseded

4.13.3 Multilingual UI

Multilingual projects MAY display the canonical terminology in the UI in the client's language (RU example):

English (canonical)	Russian (UI translation)
Business Requirement	Бизнес-требование
System Requirement	Системное требование
Test Case	Тест-кейс
Quality Gate	Контрольная точка качества
Acceptance	Приёмка
Verified	Проверено
Approved	Утверждено
Deprecated	Устарело

This is **a UI translation only**. Frontmatter, IDs, file names, and normative body paragraphs are always canonical latin / the canonical RU from this chapter.

4.14 Forbidden / deprecated terms

A closed list of non-canonical terms; the RENAR-4+ substrate hook is blocking on detection in a normative artifact (§4.2).

Forbidden term	Canonical replacement	Why
"User Story" as a requirement	`SR`	A story is a unit of planning, not a requirement; a story MAY implement an SR through `implements`
"Use Case" (formally, as an artifact)	`SPEC-UI + SR`	A use case mixes UX and behavior; RENAR separates SPEC-UI (UX) and SR (behavior)
"Spec" (as a generic term)	A concrete `SPEC-*` or `requirement` / `SR`	"Spec" is ambiguous; we use precise terms

"Business logic"	`SR`	A code term, not a requirements term
"Functionality"	`SR` / `TR`	Too broad
"Feature" (as a requirement)	`BR` (business level) or `Feature` in a SAFe context (not RENAR canonical)	Ambiguous; RENAR uses BR
"Wish-list item"	(never)	A contractual document is not written this way
"Epic" (as a requirement)	`BR` (business level)	An epic is a unit of planning, not a requirement

4.14.1 Deprecated RENAR-specific labels

When migrating from pre-v1.0 draft material, deprecated labels are encountered:

Deprecated label	Canonical v1.0 replacement
`UIC` (UI Concept)	`SPEC-UI` (§8.5.6)
`AIC` (AI Concept)	`SPEC-AI` (§8.5.7)
`TS` (Technical Specification)	`SPEC-ARCH` or `SPEC-OPS` depending on the content
`INT-SR` (Integration SR)	`SR` with `constrained-by: [SPEC-INT-N]`
`INT-TC` (Integration TC)	`TC` with `tc-type: contract`
`TM` (Module/Submodule SR)	`SR` with `level: module` (§6.7)
`QG-0 Context Gate` (old)	`QG-0 Approval Gate` (canonical v1.0, §10.3.1)
`QG-1 Requirements Gate` (old)	`QG-1 Implementation Gate` (canonical v1.0, §10.3.2) — semantic shift: previously approval of BR/SR, now only TC `draft → ready`
`QG-2 Implementation Gate` (old)	`QG-1 Implementation Gate` (canonical v1.0, §10.3.2)
`QG-3 Verification Gate` (old)	`QG-2 Verification Gate` (canonical v1.0, §10.3.3)

Migrating an existing requirements substrate with old labels is a separate one-off process; the substrate-native hook on the change-set MUST auto-detect deprecated labels and propose canonical replacements.

4.15 Order of dispute resolution (Authority)

When there is a disagreement over a term, the order of recourse is:

1. **This chapter (§4)** — canonical for the RENAR Standard.
2. **The corresponding chapter of the standard** (06–14) — for artifact-specific semantics (for example, ADAPT specifics — in §7).
3. **SENAR §3** (terminology of the parent standard).

4. **ISO/IEC 29148:2018** — for general-engineering requirements terminology.
5. **BABOK v3** — for business-analysis terms.
6. **Fixing it through the formal change procedure** of the standard (§13.9.3) — if all of the above are silent.

Do not use as a source of terminology:

- Tickets in ticket systems (often contradictory).
- Team chats (slang ≠ canonical).
- Presentations and old draft material.
- Marketing material.

4.16 Relationship to other chapters

Chapter	Relationship
02 Methodology positioning	§2.3 Source-of-Truth inversion + §2.5 substrate-independent versioning — the foundation for §4.8 substrate terms
06 Requirements hierarchy	BR / SR / TR artifact frontmatter — §4.3, §4.9, §4.12 links
07 ADAPT	ADAPT specifics — §4.3.2, §4.6.4, §4.6.6 backward sub-states
08 Specifications	SPEC-* types — §4.4, §4.6.3 SPEC lifecycle
09 Test cases	TC — §4.5, §4.6.5; pos/neg pairing — §4.5.3
10 Lifecycle and QG	Canonical Quality Gates — §4.7; state machines by type — §4.6; closed-list policy — §10.10 in parallel with §4.11 / §4.14
03 Substrate versioning	V1–V6 definitions — §4.8.2
11 Maturity model	ai-provenance mandatory at RENAR-4+ — §4.10 (criterion source — §11.7.1)
12 Metrics	Drift classes §4.11 — source for metrics such as Reconciliation Findings (§12.3.10)
13 Conformance	§13.3 mandatory clauses reference the canonical terminology of this chapter
reference/01-glossary.md	Expanded explanations, anti-patterns, history — non-normative

---

05. Roles

5.1 Who is responsible for what

RENAR does not set up its own role hierarchy — it inherits the five base roles from SENAR §4 and adds specializations for working with requirements on top of them: who owns each artifact (ADAPT, BR, SR, SPEC, TR, TC) and who is responsible for each Quality Gate. This is also where the rule for which the chapter largely exists lives: the **ACTZ dual signature** (client + implementer), without which a client decision is not an obligation. ADAPT, by contrast, is signed by the Architect alone — it is an internal interpretation.

The chapter does **not** govern the substrate-native mechanisms for implementing role checks (substrate-native ACL, V6 author identifiers, substrate-native signing events) — that is the domain of chapter 3 (substrate capabilities) and `guide/03-tool-guide-*.md` (substrate-specific tooling).

5.2 Base roles (SENAR §4)

5.2.1 Closed list

RENAR references the five base roles of SENAR §4 as the Source of Truth. The list is closed on the SENAR side; RENAR neither adds nor removes roles.

Role	Brief semantics (for the RENAR context)
Supervisor	The person who initiates and oversees the work of the AI agent; in RENAR, the typical customer for requirements-elicitation, draft-generation, and ADAPT-review tasks.
AI agent	A software participant that performs the generation and maintenance of requirement artifacts (Forward interpretation of ADAPT, BR/SR/TR/SPEC/TC, updating graph links on changes). In RENAR — the **regular primary author** of artifacts (§0.2.1); includes the primary generator and the adversarial critic. The AI agent is **not** an owner (§5.3) and does **not** place signatures (§5.5).
Architect / Tech Lead	The person normatively accountable for technical decisions and the approval of requirement artifacts (QG-0 §10.3.1); the signing party of ADAPT (§7.5) and the implementer-side party of the ACTZ dual signature (§5.5). The Source of Truth is the SENAR §4 role "Architect / Tech Lead"; this EN edition uses the short name **Architect** (the RU corpus renders it «Архитектор»).
Reviewer	A person or AI agent performing the adversarial review of artifacts; in RENAR — a mandatory role for QG-0 (§10.3.1).
Stakeholder	An external interested party — the client, an authorized client representative, a product owner, an end-user representative; the source of the TZ and the client-side signing party: ACTZ (§7.13) and QG-4 (§10.4.2).

The full definition of each role's semantics — duties, constraints, interactions — is set out in SENAR §4 and applies to RENAR unchanged.

> ***Naming of the Architect role.*** *Everywhere below in the standard, guide, and appendices, the SENAR role "Architect / Tech Lead" is denoted by the short name* ***Architect*** *(for example, "the Architect confirmed" in §8.3, the dual signature in §5.5). This is a short form, not a separate role: a conformance manifest MUST use the normative SENAR name (§5.2.2).*

> ***Naming of the Stakeholder role.*** *Everywhere below in the standard, guide, and appendices, the SENAR role "Stakeholder" is used directly as the role name (for example, "an authorized stakeholder" in §5.3.6, the client-side signing party in §5.5). It is the normative SENAR name; a conformance manifest MUST use it verbatim (§5.2.2).*

5.2.2 No override

An implementation MUST NOT:

- Rename the SENAR base roles in the normative part of the conformance manifest (§13.4).
- Merge two base roles into one in a way that removes the possibility of independent signing (for example, merging the Architect and the Stakeholder makes the ACTZ dual signature §5.5 impossible and is **not permitted**).
- Add new base roles outside SENAR §4 without the formal SENAR Standard change procedure.

Project-local **aliases** (for example, a local name "Lead Engineer" as a synonym for the SENAR Architect / Tech Lead) are permitted in communication, but the conformance manifest (§13.4) MUST use the normative role names of SENAR §4.

5.3 RENAR specializations: responsibility for artifacts

Each artifact type has a normatively fixed owner — the role responsible for initiating the artifact, its substantive integrity, and its one-click promote through QG-0 (§10.3.1).

§	Artifact	Owner	Responsibility type	QG participants
5.3.1	**ADAPT**	Architect (implementer)	Single — an internal interpretation	QG-0: Architect; QG-3 (opt.): the Architect's signature (§7.5)
5.3.1bis	**ACTZ**	Architect (implementer) + Client representative	Joint — both signatures REQUIRED	Signing (§5.5)
5.3.2	**BR / SR**	Architect	Single (Accountable); AI agent = Responsible primary generator	QG-0: Architect or authorized role-holder (§5.4); QG-2: automated runner; QG-4 (opt.): Stakeholder
5.3.3	**SPEC-***	Architect + domain technical lead	Joint by SPEC type: Architect — overall; domain TL — expertise	QG-0: Architect or authorized role-holder

5.3.4	**TR**	Architect (approval) + Engineer (execution)	Split	QG-0: Architect; QG-2: Engineer + runner
5.3.5	**TC**	Engineer + QA	Joint — Engineer authorship, QA pos/neg pairing (§9.7)	QG-1: Engineer/QA + runner; QG-2: runner with V5 pin
5.3.5bis	**AT**	Architect (implementer)	Split: authorship is isolated (§9.19.2) — an isolated AI agent on a different model generates it; the environment binding (`environment-ref`) and the run are handled on the implementer side	The AT release acceptance gate (§10.4.3): an automated runner
5.3.6	**QG-4 accepted**	Stakeholder (Client + Product Owner)	Single — Architect not sufficient	Only if QG-4 is in the manifest (§10.4.2)

5.3.1 Owner of ADAPT

ACTZ is the only RENAR artifact with **mandatory** joint responsibility: a decision does not become an obligation without the signatures of both parties (§7.13) — a normative safeguard against the unilateral fixing of obligations.

ADAPT is an artifact of the Architect's sole responsibility (§7.5): it is an internal interpretation, and the client does not confirm what they do not read. Protection against a unilateral interpretation of the TZ is provided not by a client signature on the ADAPT, but by the fact that **every finding MUST be closed by a signed ACTZ** (§7.4.5): an interpretation cannot rest on a decision the client never made.

5.3.2 Owner of BR / SR

The Architect is normatively accountable for the ADAPT → BR → SR decomposition and for the consistency of the requirements tree. The AI agent is the regular primary generator of draft BR/SR (§0.2.1, §5.2.1), but **not** the owner: the owner is always a human or an authorized role-holder (§5.4). In RACI: the AI agent is **Responsible** (executes), the Architect is **Accountable** (answers for the result). A manifest's attempt to declare the AI agent as Accountable is non-conformant (§13.3).

5.3.3 Owner of SPEC-*

The domain technical lead is the normative term for expertise on a specific SPEC type (§8.3): ARCH (system architect), API (API designer), DATA (data architect), INT (integration lead), PROC (process owner), UI (UX lead), AI (ML lead), SEC (security lead), OPS (operations lead), TEST (test-environment lead), DOC (documentation lead). In small projects one person MAY combine domain TL roles; merging the Architect + all domain TLs into one person is permitted when declared in the manifest (§13.4).

5.3.4 Owner of TR

TR has split responsibility: task approval rests with the Architect/authorized role-holder, execution with the Engineer. The Engineer cannot approve their own TR (a violation of §10.2.2).

5.3.5 Owner of TC

The AI agent is permitted as the primary generator of TC, but MUST pass a QA check before `ready` (§10.3.2). In projects without an explicit QA role, the QA participant becomes an engineer other than the author of the TC.

5.3.6 Owner of the accepted gate (QG-4)

QG-4 is the only gate for which the Architect is **not** a sufficient participant. Confirming a post-release business outcome requires an authorized stakeholder (Client + Product Owner). When QG-4 is absent from the manifest, the gate does not apply, and the Client/Product Owner role is not governed at this lifecycle stage.

5.3.7 Closed-list policy for owner assignments

The list §5.3.1–§5.3.6 is closed at v1. Project-local extensions (a new owner for a new type; reassigning owner authority between SENAR §4 roles) are **not permitted** without the formal standard change procedure (§13.9).

Negative scenario: an attempt to declare that an ACTZ is signed unilaterally (without the Client representative) is a violation of §5.3.1 and §13.3; the manifest is non-conformant. Conversely: requiring a client signature on an ADAPT is redundant and is not part of base conformance — it is a `declared-stricter` extension (§1.7.2).

5.4 Authorized role-holder

5.4.1 Normative definition

An **authorized role-holder** is a person with explicitly delegated Architect authority for a specific scope (one artifact, one artifact type, or the whole project). The delegation is recorded substrate-natively (V6 author identifier + ACL §3.3.6); the concrete mechanism is substrate-specific. "Authorized role-holder" is a normative term, identical in application across §10.2.2, §10.3.1, §13.5.

5.4.2 Permitted scenarios

A participant for the **QG-0 Approval Gate** (§10.2.2) — BR/SR/SPEC/TR/ADAPT (on the Architect side)/TC; **self-assessment** (§13.5) — an assessor with the role `authorized-role-holder` in the manifest.

5.4.3 Prohibited scenarios

An authorized role-holder does **not** replace the `client-signature` in ACTZ (the dual signature requires a client-side stakeholder, not an implementer-side authorized role-holder); does **not** replace the Stakeholder in QG-4 (§10.4.2); does **not** replace the external assessor in a third-party independent assessment (§13.6).

5.4.4 Substrate-side authorization

The delegation MUST be recorded natively via V6 (§3.3.6): a substrate with ACL — through a role-based access list; a substrate with capability tokens — through issuing a delegating token; a substrate without explicit authorization — through a declaration of delegation in a native project artifact (a separate file with the Architect's signature). Substrate-specific details — `guide/03-tool-guide-*.md`.

5.5 ACTZ dual signature

5.5.1 Normative rule

The dual signature sits on the **ACTZ** — the TZ clarification protocol (§7.13) — not on the ADAPT. An ACTZ transitions to `signed` **only** when both signatures are present:

1. `client-signature` — from the Client representative (a stakeholder authorized to sign on behalf of the client).
2. `vendor-signature` — from the implementer.

An ADAPT is signed by the Architect alone (§7.5): it is an internal interpretation artifact, and the client never sees it.

The boundary is drawn by audience: everything put to the client and approved is an obligation (ACTZ, dual signature); everything not put to the client is an interpretation (ADAPT, the Architect's signature). A client signature on an ADAPT would be a fiction — the client would be signing an engineering document they neither read nor can assess.

The structure of the signature fields is fixed in §7.13.3; each signature contains `signed-by`, `role`, `signed-at`, `signature-ref` (a substrate-native pointer to the signing event).

5.5.2 Semantics of each signature

Signature	Confirms
`client-signature` (on ACTZ)	The decisions put forward for approval are accepted; the wording of the obligations ("this is what will be built") matches the client's intent; the decisions are final.
`vendor-signature` (on ACTZ)	The implementer accepts the decisions as obligations and undertakes to implement them.
`architect-signature` (on ADAPT)	The Forward interpretation is technically feasible; all backward findings are in the `resolved` state and carry a `decided-in` pointer to a clause of a signed ACTZ (§7.4.5); the decomposition into BR / SR / SPEC is safe to launch.

The signatures are **not** interchangeable. The Architect does not confirm client semantics; the Client does not confirm technical feasibility and does not take on responsibility for the interpretation.

5.5.3 Negative scenarios

- **One signature missing on the ACTZ**: the protocol does not transition to `signed`; the decisions it carries are not obligations; findings that reference it through `decided-in` cannot be `resolved`, and the ADAPT does not transition to `approved` (§7.4.5).

- **One person in both ACTZ signatures**: an implementation where `client-signature.signed-by == vendor-signature.signed-by` violates §5.5.1 (two parties require two independent persons). The internal-product scenario without an independent client representative is outside the primary scope (§1.5.4).
- **A client signature on an ADAPT**: redundant and normatively **not required**. An implementation that demands it is a `declared-stricter` extension (§1.7.2), not base-level conformance.
- **Authorized role-holder instead of the Architect**: permitted (§5.4.2); the signature is recorded with `role: authorized-role-holder`.
- **Authorized role-holder instead of the Client**: **not permitted** (§5.4.3); the `client-signature` MUST be from a client-side stakeholder.
- **A signed decision not reflected in any interpretation**: the ACTZ is signed, but no ADAPT carries its decisions — the obligation exists outside the requirements; **fatal** (§7.4.5).

5.5.4 Relationship to other gates

The ACTZ dual signature is the only normative case of a dual signature in RENAR. The other gates (§10.3) are performed by a single participant (Architect, runner, stakeholder). This is a deliberate decision: the ACTZ is the point of agreement with the client, the other gates are internal implementation checks.

5.6 RACI matrix

The matrix fixes Responsible / Accountable / Consulted / Informed across RENAR artifact types and canonical gates. Abbreviations: **R** = Responsible (executes), **A** = Accountable (approves / answers for the result), **C** = Consulted (MUST be informed before the decision), **I** = Informed (notified after the decision).

5.6.1 Artifact matrix

Activity	AI agent	Architect	Domain technical lead	Engineer	QA	Reviewer	Client
TZ import	R	A	—	—	—	C (adversarial)	C
ADAPT creation (forward + backward)	R	A	C	—	—	C (adversarial)	C
ADAPT approval (the Architect's signature)	—	**A (`architect-signature`)**	—	—	—	C (adversarial)	I
ACTZ signing (dual signature)	—	**A (`vendor-signature`)**	—	—	—	—	**A (`client-signature`)**
ADAPT → BR decomposition	R	A	C	—	—	C	I

BR → SR decomposition	R	A	C	—	—	C	I
SPEC-* creation	R	A	**R/A (per type)**	—	—	C	I
TR creation	R	A	C	C	—	—	I
TR implementation (execution)	—	C	C	R	—	—	I
TC creation	R	C	C	R/A	A	—	I
AT generation from the effective TZ	**R (isolated agent, different model)**	A	—	—	—	—	I
Binding an AT to its environment (`environment-ref`)	—	**A**	—	R	—	—	I
Adversarial review of an artifact	R (AI critic)	A	—	—	—	R	—

5.6.2 Quality Gates matrix

Conformant to the normatively fixed participant table §10.2.2.

Gate	Artifacts	Responsible	Accountable	Consulted	Informed
QG-0 Approval Gate	BR, SR, TR, SPEC, TC, ADAPT (on the Architect side)	Architect or authorized role-holder	Architect	AI critic (Reviewer)	Team
QG-1 Implementation Gate	TC (`draft → ready`)	Engineer / QA	Engineer	Automated runner	Team
QG-2 Verification Gate	BR, SR, TR, SPEC, TC	Automated runner (V5 + V6)	Architect	—	Team
QG-3 Architecture Gate (optional, ADAPT)	ADAPT (`answered → approved`)	The Architect's signature; all findings closed by signed ACTZ	Architect	AI critic	Team
QG-4 Acceptance Gate (optional, BR)	BR (`verified → accepted`)	Stakeholder (Client + PO)	PO	Architect	Team

AT release acceptance gate (mandatory, not a QG — §10.4.3)	AT (all in `passing`); the product	Automated runner	Architect	—	Team, Client

5.6.3 Closed list of roles in RACI

The list of role columns of the matrix §5.6.1 / §5.6.2 is closed at RENAR v1:

AI agent, **Architect**, **Domain technical lead** (per SPEC type), **Engineer**, **QA**, **Reviewer** (adversarial), **Client** (authorized Stakeholder), **Product Owner**.

Project-local roles (for example, "Release Manager", "Compliance Officer") are permitted as **Informed** or **Consulted** in a local practical-RACI implementation, but do **not** appear as **Responsible** or **Accountable** in the normative conformance matrix — otherwise the manifest is non-conformant (§13.3).

5.7 Closed-list policy (closed list)

5.7.1 What is fixed at v1

The SENAR §4 base roles (§5.2.1) are closed on the SENAR side; the owner assignment §5.3.1–§5.3.6 is closed; the authorized role-holder (§5.4) is closed; the ACTZ dual-signature rule (§5.5.1) is closed; the RACI role columns (§5.6.3) are closed.

5.7.2 Declared-stricter is permitted

An implementation MAY **tighten** requirements, declaring this explicitly in the manifest (§13.4) with the `declared-stricter` marker (§10.10.2): require a dual signature for BR/SR (not only for ACTZ); require a client signature on an ADAPT; require an external adversarial reviewer at QG-0 for SPEC-SEC and SPEC-AI; prohibit combining the Architect and a domain technical lead.

5.7.3 Declared-weaker is prohibited

An implementation MUST NOT declare an ACTZ signed with a single signature; the TC author engineer approving without a QA participant; the implementer in the QG-4 participant role instead of the Stakeholder. A manifest that is declared-weaker relative to §5 is non-conformant (§13.8 loss of conformance).

5.7.4 Path for extensions

Adding a new role (§5.2, §5.3, §5.6.3) or owner combination (§5.3.7) — only through the formal standard change procedure (§13.9): research draft → public review → minor-version bump.

5.8 Cross-references

Source	Application
SENAR §4	Closed list of the 5 base roles; semantics and duties (normative source)

§7.5 ADAPT approval schema	Structure of the `architect-signature` field; `client-signature` and `vendor-signature` sit on the ACTZ (§7.13.3)
§10.2.2 QG actor table	Normative gate → participant correspondence; aligned with §5.6.2
§10.3.1 QG-0 Approval Gate	Architect or authorized role-holder as the §5.4 participant
§10.4.1 QG-3 Architecture Gate	The Architect's signature on an ADAPT (§7.5); the dual signature (§5.5) sits on the ACTZ
§10.4.2 QG-4 Acceptance Gate	Stakeholder (Client + PO) — §5.3.6
§3.3.6 V6 Author + timestamp	Substrate-native authorship recording for role-holder delegation §5.4.4
§13.4 Conformance manifest	Use of the normative role names §5.2.2
§13.5 Self-assessment	Assessor role enum (architect / authorized-role-holder / external-assessor) — §5.4 anchor
`core/renar-core.md`	Conceptual overview of the standard for the human reader (non-normative); roles and signatures are fully governed here, §5
`guide/03-tool-guide-*.md`	Substrate-specific delegation mechanisms (§5.4.4) and signature implementations

← Previous: 04. Terms and definitions · **Table of contents** · **Next: 06. Requirements hierarchy →**

06. Requirements hierarchy

> ***Part of the RENAR Standard v1.0*** · *← Table of contents*

> ***Dense chapter:*** *before the frontmatter — guide/00 quickstart; chapter density — reference/09.*

6.1 Three requirement types and three levels

A requirement has three altitudes, and they must not be conflated. **The business** wants an outcome: "the customer resets their own password to take load off support." **The system** MUST behave so that this outcome becomes possible: "on a request from a confirmed address, the system sends a reset link with a 30-minute lifetime." **The engineer** takes this on as a single task: "implement password reset with a limit of three attempts per hour." Three different questions — why, what, and exactly how — and to each RENAR assigns its own artifact type: **BR**, **SR**, **TR**.

There are exactly three types, the list is closed, and they are linked into a tree: one SR elaborates one BR, one TR elaborates one SR. Layered on top are three scale levels — system, subsystem, module; they determine which of the three types are even appropriate (a module has no business owner of its own — hence no BR). The entire Source-of-Truth hierarchy from chapter 2 §2.3 rests on this axis: TZ → ADAPT → BR / SR / SPEC → TR → TC. The structure of the system is described in parallel — by SPEC-* specifications (chapter 8), which BR / SR / TR reference through typed graph edges.

The clauses of this chapter are normative. The closed lists of types and levels are mandatory clauses (chapter 13); they can be extended only through the formal change procedure of the standard.

The chapter draws on ISO/IEC/IEEE 29148:2018 "Requirements engineering" for the concepts of business / system / task requirements and the principles of traceability, but it fixes a closed list of exactly three types on the v1.0 requirements axis and deliberately distances itself from the freely extensible set of types characteristic of classical approaches.

6.2 The closed list of three requirements-axis types

6.2.1 Normative formulation

The RENAR v1.0 requirements axis contains exactly three types: BR, SR, TR. The list is closed. New types are added only through the formal change procedure of the standard (chapter 13).

Type	Expansion	Question	Contains	Does not contain
BR	Business Requirement	Who, what, and why?	Business goal, role, value	Technologies, screens, contracts, data fields
SR	System Requirement	What does the system do?	System behavior, constraints	Table names, frameworks, concrete structures
TR	Task Requirement	What exactly to implement?	Implementation specifics: fields, conditions, errors	Architectural decisions

6.2.2 The tree of parents

```
BR
 └── SR              # parent = BR (single)
      └── TR         # parent = SR (single)
                ↓
          Goal + Acceptance Criteria
          in the task-management system
```

`SR.parent` is a single BR. `TR.parent` is a single SR. This is a **tree**, not a graph; multiple parents on the requirements axis are prohibited. The link graph runs between requirements and specifications (§6.10).

6.2.3 What is not a requirements-axis type

Artifacts historically seen in projects under the names UIC (UI Concept), AIC (AI Concept), INT-SR (integration requirement), TS (technical specification) **are not requirements-axis types in v1.0**. They have been moved to the parallel specifications axis as the corresponding SPEC types (see §8.3 for the closed list of 11 SPEC types and §8.7 for migration):

Legacy artifact	RENAR v1.0 type
UIC	SPEC-UI
AIC	SPEC-AI
INT-SR	SPEC-INT
TS (architecture / data / API / process / security / ops)	SPEC-ARCH / SPEC-DATA / SPEC-API / SPEC-PROC / SPEC-SEC / SPEC-OPS

SPEC-* relates to the requirements axis not as "one more requirement type" but as a parallel axis with typed edges `SR.constrained-by[]` and `TR.implements-spec[]` (§6.10).

Test cases (TC) are a separate class of verification artifacts, governed by chapter 9. TC are not requirements: they verify the behavior described in BR / SR / SPEC.

6.3 Systems, subsystems, modules

The levels of organizational decomposition are a closed list of three elements: **system**, **subsystem**, **module**. Each element corresponds to an allowed set of requirement types (see §6.4).

6.3.1 System

A system is the top level of the hierarchy. A whole product or platform that is delivered and operated as a single unit and for which an organization is accountable.

Indicators of a system:

- a single owner on the business side (Product Owner, director, client);
- delivered and operated as a single unit;

- the client sees and assesses the system as a whole, not its parts separately;
- a single top-level TZ document (and zero or more root ADAPTs — as many as the adversarial review produced findings for, §7.4.1.4).

Allowed requirement types: **BR, SR, TR**.

6.3.2 Subsystem

A subsystem is a large, self-contained component of the system that carries its own business value or has a separate business owner.

A subsystem is distinguished if **at least one** of the conditions holds:

Condition	Example
Its own team or technical owner	Frontend team vs AI-pipeline team
A separate database or an independently deployable service	An isolated microservice with its own deployment
The ability to be replaced independently of the others	The analytics module is replaced without changing the operational loop
A different business domain or a separate stakeholder	CFO vs COO
Added later as a separate initiative with a separate budget	A partner program

The key normative criterion: is there a separate person on the business side accountable for the value of this part of the system?

- yes → subsystem, BR are justified;
- no → module, SR only.

Allowed requirement types: **BR (if it has its own stakeholder) + SR + TR**.

6.3.3 Module

A module is a technical division within a subsystem. It implements part of the subsystem's behavior but has no business value of its own.

Indicators of a module:

- no separate business stakeholder;
- it does not exist and is not used apart from its subsystem;
- it is distinguished on a technical basis: functional area, layer, domain;
- it is not mentioned separately in the top-level TZ.

Allowed requirement types: **SR + TR**. No BR is created.

6.3.4 Summary table

	System	Subsystem	Module
Business stakeholder	yes	yes (its own)	no

Independent deployment	possible	yes	no
Mentioned separately in the TZ	yes	yes	rarely
Exists separately	yes	possible	no
BR	yes	yes (if its own stakeholder)	no
SR	yes	yes	yes
TR	yes	yes	yes

6.3.5 The "module → subsystem" evolution

The boundary between a module and a subsystem is not fixed forever. If a module has grown, gained its own team or business owner, it becomes a subsystem and gains a BR. Normatively: **a BR is written at the moment a business owner appears, not in advance**.

The reverse evolution (subsystem → module) is also permitted if the business owner has departed and the business value has become derivative; in that case the subsystem's BR is given status `deprecated` (§6.5.4), and its SR are re-parented to the parent system's BR.

6.4 Allowed requirement types by decomposition level

Level	BR	SR	TR	Explanation
System	mandatory	mandatory	mandatory	The top BR is mandatory (no business goal, no project)
Subsystem	optional	mandatory	mandatory	BR only when there is a stakeholder of its own
Module	not allowed	mandatory	mandatory	BR is normatively prohibited

The normative rationale for prohibiting BR at the module level: a business requirement without a business stakeholder and without independent business value leads to false decomposition and breeds "technical BR" indistinguishable from SR. This blurs the Source-of-Truth hierarchy (chapter 2 §2.3).

6.5 BR — Business Requirement

6.5.1 Normative definition

A BR records **what the business needs and why**. It describes the role, the action, and the business value without references to technologies, screens, contracts, or data structures.

6.5.2 BR frontmatter (mandatory fields)

```
---
id: BR-NN                          # immutable; NN sequential within scope
```

```yaml
title: "<short, descriptive>"
type: BR
slug: "<kebab-case>"                # auto-derived

# === Scope (mandatory) ===
level: system | subsystem           # BR at the module level is prohibited
(§6.4)
scope:
  system: "<system-id>"
  subsystem: "<subsystem-id>"       # null if level=system

# === Lifecycle (mandatory) ===
status: draft | approved | verified | deprecated
owner: "<role / responsible person>"

# === Source: provenance (conditional, see §7.4.1) ===
# source.adapt — mandatory if an ADAPT exists (a gap was found during TZ → RENAR
conversion);
# source.tz-section — always mandatory. At least one source is always present.
source:
  adapt: ADAPT-NNN                  # conditional: present if an ADAPT was
created for this TZ
  adapt-section: "Forward §N"       # mandatory if adapt is present
  tz-section: "§N.N"                # always mandatory — primary provenance
  adversarial-review-ref: AR-NNN   # conditional: present if source.adapt is
absent — AR with the "no findings" verdict (§7.4.6)

# === Cross-level link from subsystem BR → system BR (see §6.8.2) ===
# Recommended on v1.0; mandatory on v1.1+ when level=subsystem AND the parent
system has an approved BR.
# Not a parent edge — a separate link-graph edge type (see §6.8.3).
implements:                         # array; substrate-agnostic
  - id: BR-NN                       # ID of the parent system's BR
    scope:
      system: "<system-id>"         # mandatory for a cross-system reference
    rationale: "<short>"            # optional; reference to ADAPT§ if available

# === Link graph (auto-managed) ===
children: []                        # auto-derived; SR referencing parent.id=
<this BR>
implemented-by: []                  # auto-derived; subsystem BR referencing
implements[].id=<this BR>
verified-by: []                     # auto-derived; TC verifying through SR

# === AI provenance (mandatory on RENAR-4+; canonical schema — §4.10.1) ===
ai-provenance:
  generated-by: "<vendor>-<model>@<date>"
  generated-at: "<ISO-8601>"
  prompt-template: "<template-path>@<version>"
  context-tokens: integer
  output-tokens: integer
  human-edits: boolean
  # optional on RENAR-4, mandatory on RENAR-5 (see §4.10.1):
  # cost-budget, cost-actual, generation-time-ms
```

```
# === Replacement (mandatory if applicable) ===
replaces: "<old-id>"
replaced-by: "<new-id>"
deprecated-date: "<ISO date>"
---
```

The field `source.tz-section` is always mandatory. The field `source.adapt` is conditional: it is present when the TZ → RENAR conversion required an ADAPT (§7.4.1.1), and is omitted when the adversarial reviewer returned a "no findings, no clarifications" verdict (§7.4.1.2). If `source.adapt` is omitted, the field `source.adversarial-review-ref` is mandatory: it points to the adversarial-review record AR (§7.4.6) that holds that verdict for audit. The lifecycle hooks (§7.4.1, §10.11.1) check both cases: (1) if `source.adapt` is present — that the ADAPT is in status `approved` or higher; (2) if `source.adapt` is omitted — that `source.adversarial-review-ref` points to an existing AR in status `issued` with the `no-findings` verdict.

The field `parent` is absent in a BR: a BR is the root node of the requirements tree. For the cross-level link between a subsystem tree and a system tree, the separate field `implements[]` is used (see §6.8.2): this is **not** a parent edge but a typed cross-level declaration "this subsystem BR elaborates the listed system BR." The prohibition on multiple parents (§6.8.3) does not apply to `implements[]`.

6.5.3 BR body (mandatory sections)

Section	Obligation	Content
Need	mandatory	Who (role), what (action), why (business goal). Stated in one sentence.
Success criteria	mandatory	Measurable outcomes (3–7 items); each independently verifiable.
Context	mandatory	Where the requirement came from (with a reference to an ADAPT section), what alternatives were considered.
Constraints	optional	Business constraints (budget, deadlines, regulation), not technical ones.

Technical detail (UI, API, data model) is prohibited in a BR — the SPEC types (chapter 8) and SR exist for that.

6.5.4 BR statuses

Status	Meaning	Transition trigger
`draft`	In progress	Created by the author
`approved`	Approved, may be decomposed into SR	After QG-0 (chapter 10)
`verified`	All derived SR / TR / TC are complete, the business outcome is confirmed	After QG-2; all `verified-by` TC have `last-run.result = pass` on the current version

`deprecated`	Obsolete; superseded by another BR or no longer relevant	By the Architect / Product Owner, mandatorily with `replaced-by` (if there is a replacement)

A BR in status `deprecated` is **not deleted** — it remains as a historical trace for audit.

6.6 SR — System Requirement

6.6.1 Normative definition

An SR records **what the system does** (at the system, subsystem, or module level). It describes observable behavior and constraints. It does not describe table names, frameworks, or concrete data structures — that is the responsibility of SPEC (chapter 8).

6.6.2 SR frontmatter (mandatory fields)

```yaml
---
id: SR-NN                              # immutable
title: "<short, descriptive>"
type: SR
slug: "<kebab-case>"

# === Scope (mandatory) ===
level: system | subsystem | module
scope:
  system: "<system-id>"
  subsystem: "<subsystem-id>"          # null if level=system
  module: "<module-id>"                # null if level ≠ module

# === Lifecycle (mandatory) ===
status: draft | approved | verified | deprecated
owner: "<role / responsible person>"

# === Parent (mandatory) ===
parent:
  id: BR-NN                            # single parent

# === Source: provenance (conditional, see §7.4.1) ===
# Same rules as for BR: source.adapt conditional; source.tz-section always
mandatory;
# source.adversarial-review-ref mandatory if source.adapt is omitted.
source:
  adapt: ADAPT-NNN                     # conditional
  adapt-section: "Forward §N"          # mandatory if adapt is present
  tz-section: "§N.N"                   # always mandatory
  adversarial-review-ref: AR-NNN   # mandatory if adapt is omitted (§7.4.6)

# === Quality characteristic (conditional) ===
# Mandatory for a non-functional SR; the value comes from the ISO/IEC 25010:2023
# list (§14.4.3). For a functional SR the field is omitted.
```

```yaml
quality-characteristic: functional-suitability | performance-efficiency |
compatibility |
                        interaction-capability | reliability | security |
maintainability |
                        flexibility | safety

# === Link graph (mandatory ones + auto-managed) ===
constrained-by:                      # typed edges to SPEC (chapter 8)
  - SPEC-UI-NN
  - SPEC-API-NN
  - SPEC-DATA-NN
children: []                         # auto-derived; TR referencing parent.id=
<this SR>
verified-by: []                      # auto-derived; TC verifying the SR

# === AI provenance (mandatory on RENAR-4+) ===
ai-provenance:
  generated-by: "<vendor>-<model>@<date>"
  prompt-template: "<template-path>@<version>"
  context-tokens: integer
  output-tokens: integer
  human-edits: boolean

# === Replacement (mandatory if applicable) ===
replaces: "<old-id>"
replaced-by: "<new-id>"
deprecated-date: "<ISO date>"
---
```

Key fields. `parent.id` is a single BR; this is a tree of parents. `constrained-by[]` are typed references to SPEC-*; this is a **graph**, not a tree. An SR may reference any number of SPEC of any types; a SPEC, in turn, may be `referenced-by` many SR (chapter 8 §8.2).

6.6.3 SR body (mandatory sections)

Section	Obligation	Content
Requirement	mandatory	One sentence in normative form: "The system MUST ..." (modality per the convention of §0.5: "MUST" / "SHALL" = mandatory).
Behavior	mandatory	A detailed description of observable behavior; functional scenarios.
Constraints	mandatory if applicable	Non-functional constraints (performance, security); full constraints live in the `constrained-by[]` SPEC.
Link to SPEC	mandatory if `constrained-by[]` is present	A short explanation of which aspects of behavior are governed by which SPEC.

6.6.4 SR statuses

Identical to BR statuses (§6.5.4) — `draft → approved → verified → deprecated`. The `approved → verified` transition is after QG-2 (chapter 10); it requires all `verified-by` TC to have `last-run.result = pass` on the current SR version.

6.7 TR — Task Requirement

6.7.1 Normative definition

A TR is the atomic unit of an implementer's work. It records **what exactly to implement** within a single SR. A TR decomposes an SR down to a level fit for direct implementation (one task — one TR).

6.7.2 TR frontmatter (mandatory fields)

```
---
id: TR-NN                              # immutable
title: "<short, descriptive>"
type: TR
slug: "<kebab-case>"

# === Scope (mandatory) ===
level: system | subsystem | module   # system — rare, cross-subsystem tasks
scope:
  system: "<system-id>"
  subsystem: "<subsystem-id>"        # null if level=system
  module: "<module-id>"              # null if level ≠ module

# === Lifecycle (mandatory) ===
status: draft | approved | done | obsolete
owner: "<assignee role / agent>"

# === Parent (mandatory) ===
parent:
  id: SR-NN                          # single parent

# === Source: trace chain (auto-derived from parent SR) ===
# A TR has no source of its own — it inherits from the parent SR (§6.7.5).
# If parent SR.source.adapt is omitted (§7.4.1), that fact is inherited too.
source:
  adapt: ADAPT-NNN                   # auto-derived from parent SR; may be
omitted
  sr-version: "<version-ref>"        # pinning to the SR version (substrate
capability V5; chapter 3)

# === Link graph ===
implements-spec:                     # typed edges to SPEC
  - SPEC-API-NN
  - SPEC-UI-NN
```

```yaml
verified-by: []                       # auto-derived; TC verifying through SR

# === Goal + Acceptance Criteria ===
goal: "<one-sentence outcome>"
acceptance-criteria:
  - "<numbered, falsifiable, unambiguous>"
  - "..."

# === AI provenance (mandatory on RENAR-4+) ===
ai-provenance:
  generated-by: "<vendor>-<model>@<date>"
  human-edits: boolean
---
```

Key fields. `parent.id` is a single SR. `implements-spec[]` are typed edges to SPEC; they specify which SPEC must be taken into account when implementing this particular TR (a subset of the parent SR's `constrained-by[]` or its extension with SPEC types not listed directly on the SR). `acceptance-criteria` is a closed, numbered list of falsifiable statements.

6.7.3 TR body (mandatory sections)

Section	Obligation	Content
Goal	mandatory	One paragraph; the outcome that the TR makes observable.
Acceptance Criteria	mandatory	A numbered list; each item falsifiable; covers positive and negative scenarios.
Scope	mandatory	What is in / out of the TR (matches SENAR Rule 2).
References	mandatory if applicable	To SPEC from `implements-spec[]` and to sections of the parent SR.

6.7.4 TR statuses

Status	Meaning	Trigger
`draft`	TR created, AC not yet finalized	Authoring
`approved`	AC approved, work may start	QG-0 (chapter 10): goal + AC present
`done`	AC verified, TC passing	QG-2; `verified-by` TC `pass`
`obsolete`	TR no longer relevant before completion (e.g. the SR changed)	By the Architect, mandatorily with a note

6.7.5 A TR does not reference ADAPT directly

The implementer of a TR works within the SR / SPEC and **does not turn to ADAPT directly** — all the needed interpretations of the TZ are already recorded in the approved ADAPT and threaded into SR / SPEC

through `source.adapt` (chapter 7 §7.7.3). If the implementer finds an ambiguity in the SR, this is a signal either for a new Backward finding in ADAPT (if the root of the ambiguity is in the TZ) or for a clarification of the SR (if the root is in the decomposition).

6.8 Extended hierarchy for composite systems

The base scheme `BR → SR → TR` holds for most projects. For composite systems the standard governs two variants of the extended hierarchy.

6.8.1 Subsystem as a technical division, not a standalone product

BR pertain to the system as a whole; subsystems are a technical division by teams, components, or other architectural boundaries:

```
BR (system)
 └── SR (system)          # optional, if there are cross-subsystem SR
      └── SR (subsystem)
           └── TR

SPEC-INT (between subsystems)    # parallel axis; see chapter 8
```

`SPEC-INT` belongs to none of the subsystems — it is a system-level integration specification.

6.8.2 Subsystem as a standalone product with its own stakeholder

The subsystem has its own BR with its own business owner:

```
BR (system)
 └┄┄ BR (subsystem)       # ┄┄ implements-edge (see below), NOT a parent edge
      └── SR (subsystem)
           └── TR
```

`└┄┄` between BR (system) and BR (subsystem) denotes a **typed cross-level `implements` edge** (§6.5.2): the subsystem BR declares which BR of the parent system it elaborates and implements. This is **not a parent edge** of the requirements tree: `BR (subsystem).parent` remains absent, and each such subsystem is the root node of its own tree. `implements` is a separate link-graph edge type, symmetric to the `constrained-by[] ⇔ referenced-by[]` pair for SPEC (§6.10.2).

The normative rule for `implements[]`:

Level	Scenario	Rule
Recommended on v1.0 / mandatory on v1.1+	`BR.level = subsystem` AND `scope.system` has ≥1 approved BR	`implements[]` MUST contain ≥1 reference to an applicable BR of the parent system

Permitted	`BR.level = subsystem` AND the parent system is a container with no BR of its own (organizational-level scope)	`implements[]` is omitted; the rationale is recorded in the `Context` section with a reference to ADAPT§
Prohibited	`BR.level = system`	`implements[]` does not apply (a system BR is the root of the whole scope hierarchy)

The lifecycle hooks (chapter 10 §10.11) MUST:

- Check that the target BR exists (by `id + scope.system`) when approving a subsystem BR.
- Check that the target BR is in status `approved` or higher; an erroneous draft target is fatal.
- Detect cycles in `implements` chains; a cycle is fatal.
- On deprecating a target BR (§6.5.4) — generate a cascade-warning for all `implemented-by[]` (not a cascade-deprecate; the decision on the evolution of the dependent BR rests with the Architect).

The machine-readable trace chain in §6.8.2 is reconstructed through the `implements` edge (§6.10.3) — the asymmetry with §6.8.1 is removed.

The subsystem's link to the system's shared ADAPT is preserved through `source.adapt` (if applicable); `implements[]` and `source.adapt` are independent fields and may point to different nodes of the graph.

6.8.3 The prohibition on multiple parents

The standard does not allow multiple `parent` for an SR or TR. Cross-functional requirements that might look like "children of two SR at once" are governed in one of two ways:

- they are split into several SR, each with a single parent BR;
- they are decomposed into a higher-level SR (the parent subsystem or system) on which these cross-functional scenarios depend.

The field `BR.implements[]` (§6.5.2, §6.8.2) is **not a parent edge** and is not subject to §6.8.3: a single subsystem BR may elaborate several system BR (cardinality 0..N). This is a deliberate difference in the typing of link-graph edges: parent is single, cross-level declarations are multiple.

6.9 Evolution of the hierarchy

6.9.1 Module → subsystem

The scenario from §6.3.5: a module gains a business owner. The normative sequence:

1. The appearance of a business owner is recorded through a Backward finding in ADAPT (category `scope` — a change of work boundaries; chapter 7 §7.4.4).
2. After the delta-ADAPT is approved — the module is promoted to subsystem status; a subsystem BR is created with `source.adapt: <delta-ADAPT>`.
3. The module's existing SR are preserved (immutable IDs); the SR `parent` field is updated to the new subsystem BR in an atomic change.

4. TR / TC referencing these SR require no changes (the parent SR is unchanged).

6.9.2 The prohibition on anticipatory hierarchy

Creating a subsystem BR "for growth," without an existing business owner, is a violation of the standard. A BR without a stakeholder turns into a "technical BR" that blurs the Source-of-Truth inversion and substitutes for an SR. The lifecycle hooks (chapter 10) MUST block the transition of a BR to `approved` if no identified business owner is recorded in ADAPT.

6.9.3 Subsystem → module

The symmetric scenario of §6.3.5: the subsystem has lost its business owner or the business value has become derivative of the parent system. The normative sequence:

1. The loss of the business owner / the reassessment of business value is recorded through a Backward finding in ADAPT (category `scope`; chapter 7 §7.4.4).
2. After the delta-ADAPT is approved — the subsystem BR is moved to status `deprecated` with the reason given in `Context` (business owner withdrawn / business value absorbed by the system). The BR is not deleted (immutable IDs, V1).
3. The subsystem's SR are preserved (immutable IDs); the SR `parent` field is updated in an atomic change to the parent system's BR (or to another subsystem's BR, if the SR's area belongs to it).
4. The subsystem is renamed to a module at the level of the area and the storage scheme (§6.11.2); the existing SR / TR IDs remain unchanged.
5. TR / TC referencing these SR require no changes.

If no BR of the parent system covers the behavior of an SR, this is a signal that the subsystem has not in fact lost its independent business value; reverse evolution is impossible in that case, and the subsystem BR remains `approved`.

6.10 Linking the hierarchy to ADAPT and SPEC

The standard fixes the links between the requirements axis, ADAPT, and the parallel SPEC axis through normative frontmatter fields and typed link-graph edges.

6.10.1 Link to ADAPT

`BR.source.adapt`, `SR.source.adapt`, `SPEC-*.source.adapt` are conditional references to the ADAPT from which the artifact was derived (chapter 7 §7.7.1): present when an ADAPT was created; on a "no findings" verdict no ADAPT exists, and instead of the reference the field `source.adversarial-review-ref` is mandatory (§7.4.1). A TR has no direct `source.adapt` — it inherits it through the `parent SR` (§6.7.5).

6.10.2 The link graph with SPEC

```
Requirements tree (behavior axis):       Parallel specifications axis:
BR
 └── SR  ─── constrained-by[] ──►    SPEC-ARCH  SPEC-API  SPEC-DATA
```

```
      └── TR ─── implements-spec[] ──►   SPEC-INT   SPEC-PROC SPEC-UI
                                          SPEC-AI    SPEC-SEC  SPEC-OPS
                                          SPEC-TEST  SPEC-DOC
```

Edge	Type	Obligation
`SR.constrained-by[] → SPEC-*`	Graph (multiple)	Optional; present when governing SPEC exist
`TR.implements-spec[] → SPEC-*`	Graph (multiple)	Optional; specifies SPEC for the implementer
`SPEC-*.referenced-by[] → SR / TR`	Auto-derived inverse	Auto-computed by substrate-native indexing
`SPEC-*.depends-on[] → SPEC-*`	Graph between SPEC	See chapter 8 §8.2

The link between the requirements axis and the SPEC axis is normative: an SR with non-trivial UI / API / data behavior MUST NOT remain without `constrained-by[]` (its absence is a signal either to write a SPEC or to justify the absence in the SR's Context).

6.10.3 The full trace chain (read-side)

ADAPT is a reactive artifact (§7.4.1): it is created only when the TZ → RENAR conversion produces a gap between the languages. The trace chain accordingly has **two valid variants**, chosen depending on whether an ADAPT exists for the specific TZ.

Variant A — when an ADAPT was created (`source.adapt` present):

```
TC-NN  →  verifies SR-12 v1.4
              │
              ├─ parent:        BR-03 v2.0     (BR-03 — level: subsystem)
              │                     │
              │                     └─ implements: BR-01 (system), BR-05 (system)
              │                                          (typed cross-level edge
§6.8.2)
              ├─ source.adapt:  ADAPT-001 §Forward §3.2
              │     └─ source-tz: TZ-2026-001 §3.4
              ├─ constrained-by: SPEC-UI-04, SPEC-API-02
              └─ children:      TR-101, TR-102, TR-103
                                    └─ implements-spec: SPEC-API-02
                                    └─ verified-by: TC-NN
```

Variant B — when no ADAPT was created (`source.adapt` omitted; the adversarial reviewer returned a "no findings" verdict, §7.4.1.2):

```
TC-NN  →  verifies SR-12 v1.4
              │
              ├─ parent:        BR-03 v2.0     (BR-03 — level: subsystem)
```

```
                    │                       │
                    │                       ├─ implements: BR-01 (system), BR-05 (system)
                    │                       └─ source.tz-section: TZ-2026-001 §3.4
                    │                          source.adversarial-review-ref: AR-002
                    ├─ source.tz-section: TZ-2026-001 §3.4   (no ADAPT for this TZ)
                    │  source.adversarial-review-ref: AR-002
                    ├─ constrained-by: SPEC-UI-04, SPEC-API-02
                    └─ children:      TR-101, TR-102, TR-103
                                           └─ implements-spec: SPEC-API-02
                                           └─ verified-by: TC-NN
```

Both variants are **machine-readable**. In variant B the path is reconstructed through `source.tz-section` directly; the `adversarial-review-ref` reference points at the adversarial-review record AR (§7.4.6), which records who declared "no findings" and when (V6 author + timestamp), and is available on an auditor's request (§13.5).

For the "subsystem as a standalone product" scenario (§6.8.2) the chain in both variants contains the `implements` edge between the subsystem BR and the parent system BR — this reconstructs machine-readable traceability symmetric to the §6.8.1 scenario.

A superseded ADAPT in the trace chain. When an ADAPT moves to `superseded` (§7.6.4, §10.8.5), all derived BR / SR / SPEC with `source.adapt` pointing to it MUST be either redirected to the superseding ADAPT or re-derived. A dangling `source.adapt` reference to an ADAPT in status `superseded` makes the trace chain **invalid** (read-side): an audit must not lead to a superseded source of interpretation as if it were in force. The `superseded` ADAPT itself is preserved for audit (V1) and is reachable through the `superseded-by` edge from the superseding ADAPT — but not as the `source.adapt` of an in-force requirement. Enforcement is the `adapt-supersession` validation (§10.11.1).

6.11 Storage scheme

Requirements are stored in subfolders of the requirements substrate. The substrate-native storage implementation is substrate-specific (see guide/03 for distributed VCS; guide/04 for a document-oriented store).

6.11.1 At the system level

```
[requirements-substrate]/        # root of the requirements substrate (layout —
guide/03 or guide/04)
  br/                            # BR-NN-*.md
  sr/                            # SR-NN-*.md, level=system
  tr/                            # TR-NN-*.md, level=system (rare)
  specs/                         # SPEC-* (chapter 8)
  adapt/                         # ADAPT (chapter 7)
  tz/                            # immutable TZ sources
  REQUIREMENTS.md                # auto-generated index
```

6.11.2 At the subsystem / module level

```
[subsystem-substrate]/           # subsystem scope within the requirements
substrate
  br/                            # if the subsystem has its own stakeholder
  sr/
  tr/
  modules/
    [module-substrate]/
      sr/                        # a module has only SR + TR
      tr/
  specs/                         # chapter 8
  adapt/
  REQUIREMENTS.md
```

6.11.3 REQUIREMENTS.md — auto-generated index

`REQUIREMENTS.md` is an auto-generated registry of all BR / SR / TR in the area: ID, type, level, title, status, parent, link to file. It is marked with a substrate-native auto-generated flag. Regeneration triggers are every frontmatter change or every approve / verify gate (chapter 10).

6.12 Quality Gates for the requirements axis

Detailed gate definitions are in chapter 10. A brief summary for BR / SR / TR:

Gate	Applies to	Precondition	Postcondition
QG-0 (Approval)	BR / SR / TR (`draft → approved`)	frontmatter valid, mandatory fields filled, identifier unique (V1); `source.adapt`, if present, points to an approved ADAPT, otherwise `source.adversarial-review-ref` is present (BR / SR); `parent` points to an approved BR / SR (SR); body sections conform to §6.5.3 / §6.6.3; adversarial review performed	The artifact is moved to `approved`; immutable until the next change's version; decomposition into child artifacts is allowed
QG-2 (Verification)	BR / SR / TR (`approved → verified` / `done`)	All `verified-by` TC `pass` on the current artifact version	The artifact is `verified` (BR/SR) or `done` (TR); the chain up to the parent BR is updated to verified if all children are verified

QG-1 (Implementation) **does not apply** to the requirements axis: it is a separate gate for TC only (§10.3.2) — there is no intermediate "QG-1 implementation" for BR / SR / TR; the `approved → verified` / `done` transition is governed by the single QG-2. A TR transitions `draft → approved` through the same QG-0 in a single step (frontmatter + goal + AC). `ready` and similar terms are not requirements-

axis statuses and do not appear in the state machine `draft → approved → verified | done | obsolete | deprecated` (see §6.5.4 / §6.6.4 / §6.7.4).

The substrate hooks (chapter 3 §3.3) MUST block transitions that violate the precondition of the corresponding gate.

6.13 Implementation-originated requirement (`implementation-originated`)

6.13.1 The normative rule

The Source-of-Truth inversion (§2.3.3) forbids deriving a requirement from the observed behavior of the code and retrofitting an SR to what has already been written. The prohibition is **not relaxed**: it is part of MVR-1.

But the prohibition has a price it must not charge. An AI agent implementing a task routinely adds **internal technical details** that are absent from the requirements: a defensive input check, an edge-case handler, logging. Demanding a delta-ADAPT with a client signature for every such detail is an absurd ceremony — and it is precisely that ceremony which pushes teams toward silently retrofitting requirements after the fact.

Hence a **narrow** legal class is introduced: `source: implementation-originated` .

6.13.2 The boundary of the class

What	Where it goes
Behavior observable by the client (it affects acceptance: a screen, a field, a message, a deadline, a format, the scope of delivery)	**Only** through the contractual contour: ADAPT / ACTZ with signatures. There are no concessions
An internal technical detail with no client-observable behavior (a defensive check, an internal validation, logging, an edge-case handler)	`source: implementation-originated` is permitted

The boundary is defined by **observability to the client**, not by the size of the change. Doubt is resolved in favor of the contractual contour.

6.13.3 The mandatory envelope

A requirement of the `implementation-originated` class MUST carry:

1. **Provenance**: a reference to the implementation change unit that gave rise to it, and a rationale — why the functionality was deemed necessary.
2. **Human approval**: an explicit approval by a human Supervisor (a human, not an agent). An agent MUST NOT legalize its own addition.
3. **A TC before merge**: a covering test MUST exist and pass before the change is merged.
4. **Mutation checking** (§9.18.2): such a TC **cannot have a red history** — it is written after the code and is born green. Its teeth are proven by a **killed mutant**; without a killed mutant the TC is not evidence.

5. **A counter in the drift metrics** (§12.3): a growing share of `implementation-originated` is a signal that the requirements process is leaking, and that signal MUST be visible.

6.13.4 What remains forbidden

- Silently editing an existing SR to match the observed behavior of the code remains a **violation** (§2.3.3, MVR-1). The `implementation-originated` class legalizes **adding** an internal detail with provenance and a human signature — not **rewriting** a norm to fit a fact.
- The `implementation-originated` class applied to client-observable behavior is **non-conformance**: an obligation toward the client cannot originate in the code.

6.14 Links to other chapters

Chapter	Link
02 Positioning in the methodology typology	The BR / SR / TR hierarchy is the load-bearing structure of the Source-of-Truth inversion (Claim 1); the waterfall form (Claim 2) sets the BR → SR → TR layers
07 ADAPT	`BR.source.adapt`, `SR.source.adapt` are conditional (when there is no ADAPT — `source.adversarial-review-ref`); the requirements axis is derived from an approved ADAPT or directly from the TZ on a "no findings" verdict
08 Specifications	The parallel SPEC axis; `SR.constrained-by[]`, `TR.implements-spec[]` are typed graph edges
09 Test cases	TC verify BR / SR / TR; `verified-by[]` is an auto-derived inverse
10 Lifecycle and QG	The BR / SR / TR state machines; QG-0 / QG-2 for the requirements axis (QG-1 — for TC only)
03 Substrate versioning	Immutable IDs (V1); atomic change unit on re-parenting (V2); diff & review for approve (V3); versioning without loss of history (V4); pinning SR-version in TR (V5); substrate-native approve signature with author + timestamp (V6)
11 Maturity model	RENAR-1: the BR / SR / TR axis is mandatory; RENAR-3+: `constrained-by[]` for all SR where applicable
13 Conformance	The closed list of types (BR / SR / TR) is a v1.0 mandatory clause; the closed list of levels (system / subsystem / module) is a v1.0 mandatory clause
reference/02 — schemas	The full machine-readable schema of BR / SR / TR frontmatter

---

07. ADAPT — two-way TZ adaptation

7.1 What ADAPT is and why it exists

The client sends a TZ in their own language — the language of business and of the contract. It is signed and no longer changes: it is a contract. But turning it directly into precise requirements almost never works. Somewhere the TZ is silent about something important ("data export" — in what format? for what retention period?), somewhere it contradicts itself, somewhere the same term means one thing to the client and another to the engineer. Editing the TZ is not allowed — it is a contract. Silently filling in the gaps on the client's behalf is also not allowed: that way someone else's guesses seep into the requirements, and at acceptance the "this is not what we ordered" surfaces.

ADAPT is the bridge across this gap. It has two sides: **forward interpretation** ("we understood §4.2 of the TZ as follows") and **backward findings** ("§4.3 does not set a deadline — please clarify"), which go to the client and come back as the client's decisions.

But the document the client reads and the document the engineer works from are **two different documents**. Decisions put to the client and signed are obligations, and they live in a separate artifact: **ACTZ**, the TZ Clarification Protocol (§7.13). ADAPT remains an internal interpretation: only the Architect signs it, and the client never sees it. The boundary is drawn not by the content of a record but **by audience** — everything shown to the client and approved is an obligation; everything not shown is an interpretation. The intractable argument "is this an interpretation or already a change?" thereby disappears.

ADAPT need not precede all derivation. The typical occasion to create it is TZ import, but a question to the client rooted in the TZ may also surface **later** — during the BR → SR → SPEC decomposition, or while developing test cases. A **new** ADAPT then arises, bound to the stage where the finding was discovered, while previously derived artifacts remain valid. ADAPT is not always created: if the conversion of the relevant TZ fragment is unambiguous and there are no questions, it is not needed (§7.4.1).

ADAPT is a consequence of Statement 2 of §2.4 (RENAR is a waterfall-form ≠ classical waterfall, because ADAPT provides two-way adaptation instead of "throwing the specification over the wall").

7.2 The problem ADAPT regulates

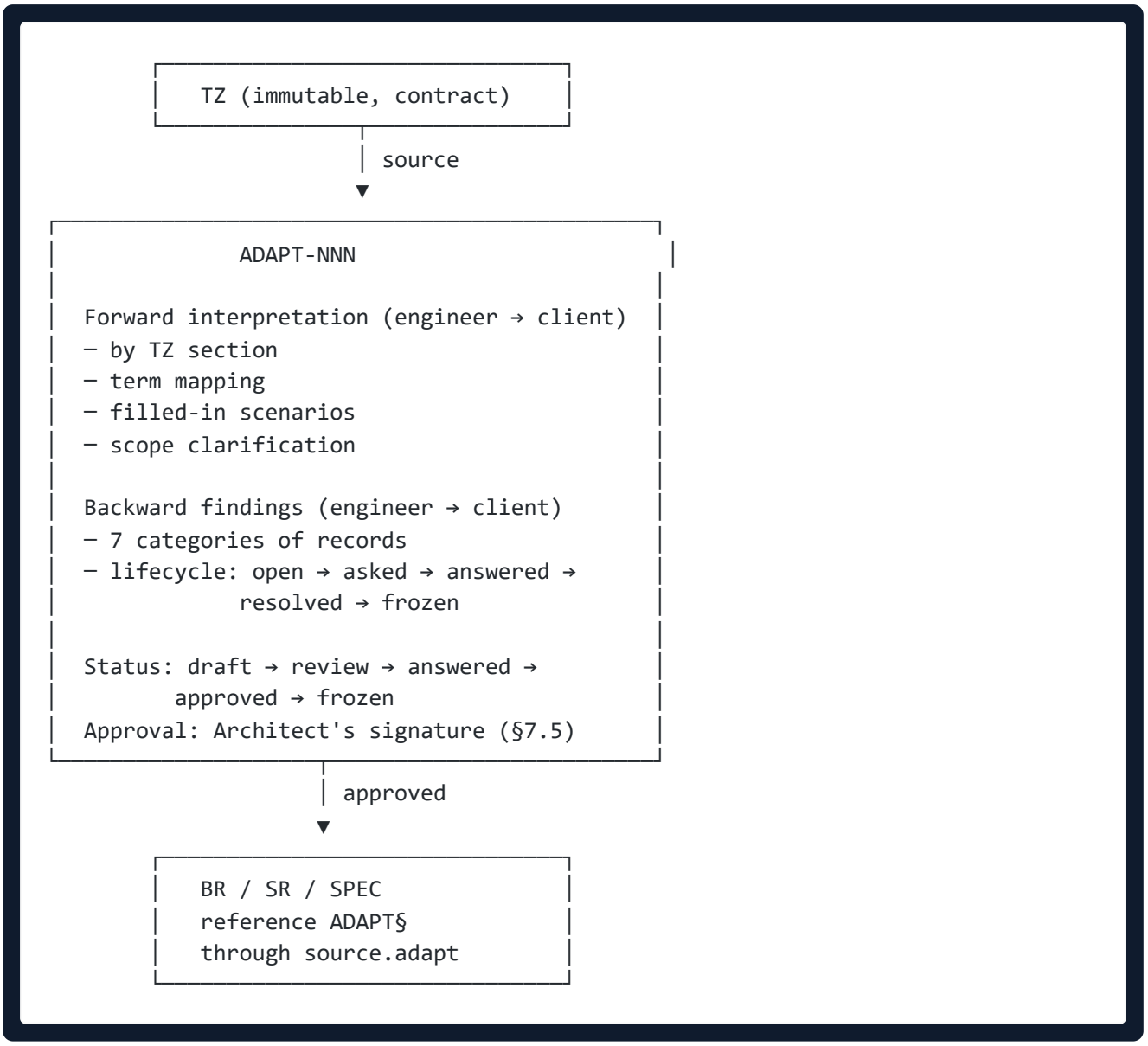
Without a formalized intermediate artifact between the TZ and BR/SR, one of two negative outcomes arises:

Outcome	What happens
TZ drift	The TZ is edited after signing → the contract is breached
Hidden interpretation	Engineering assumptions silently seep into BR/SR/SPEC → the trace chain and provenance break down

ADAPT eliminates both outcomes: the TZ remains an immutable contractual document, **all** interpretations, clarifications, and backward findings are registered in ADAPT, which itself becomes immutable after the

Architect's signature (§7.5), while the client's decisions are registered in ACTZ protocols signed by both parties (§7.13).

7.3 The two-way adaptation cycle



The TZ is the primary source; ADAPT is the canonical source of interpretation; BR/SR/SPEC reference ADAPT, not the TZ directly.

The "TZ → ADAPT" input shown is the typical one (TZ import), but **not the only one**. The cycle trigger is stage-agnostic: if a question to the client rooted in the TZ surfaces later — at the BR → SR → SPEC decomposition stage or while developing TC — the cycle runs again and produces a **new** ADAPT-NNN, bound to the stage at which the finding was discovered (§7.4.1.1). A single TZ may have several such ADAPTs (MVR-3, §0.5).

7.4 Normative requirements for ADAPT

7.4.1 Reactive obligatoriness

ADAPT is a **reactive artifact**: it is created if and only if converting the TZ → RENAR description produces a gap between the client's language and the requirements language (§7.12). If the conversion is unambiguous, no ADAPT is created; BR / SR / SPEC are derived directly from the TZ through the mandatory `source.tz-section` field.

7.4.1.1 Conditions for ADAPT obligatoriness

ADAPT is REQUIRED **if and only if** at least one of the conditions holds:

1. **A backward finding is discovered.** The adversarial reviewer (§7.10.2) at any stage of the requirements lifecycle (TZ import or BR → SR → SPEC → TC decomposition) has identified ≥ 1 record under at least one of the 7 categories of §7.4.4 (`contradiction` , `gap` , `hidden-assumption` , `feasibility` , `regulatory` , `terminology` , `scope`) rooted in the language or intent of the TZ.
2. **Term mapping is required.** The TZ uses a term that has no unambiguous engineering interpretation (requires a "client → engineer" mapping).
3. **Scope clarification is required.** The scope from the TZ is ambiguous and requires fixing "in / out".

If none of the conditions holds (the adversarial reviewer returned a "no findings, no clarifications" verdict), no ADAPT **is created**. BR / SR / SPEC reference the TZ directly through `source.tz-section` (§6.5.2, §6.6.2, §8.5).

7.4.1.2 The adversarial reviewer as a mandatory gate

The adversarial review of the TZ (§7.10.2) is a **mandatory step** for every TZ at import, regardless of whether ADAPT is created or not. The verdict at import is one-time but **not final**: at the derivation stages (BR → SR → SPEC → TC) the adversarial reviewer issues the verdict again as decomposition uncovers new questions about the TZ. A "no findings" verdict at import does not block the appearance of ADAPT later, if a finding rooted in the TZ is discovered at the decomposition stage (§7.7.3). The adversarial reviewer issues a formal verdict in one of two forms:

Verdict	What is recorded	Consequence
"findings present"	A list of concrete findings across the 7 categories + an indication of TZ sections	ADAPT is REQUIRED; the lifecycle of §7.4.5 starts
"no findings, no clarifications"	Confirmation by the adversarial reviewer (a different model, §9.4) that the TZ converts to RENAR unambiguously	ADAPT MAY be omitted; the verdict remains a recorded piece of evidence

In both outcomes the verdict MUST be recorded in an **adversarial-review record** (AR, §7.4.6) — an evidence record, immutable once issued, carrying its own identifier, author and timestamp (V6). AR is the sole carrier of the verdict: on the "findings present" outcome it points to the ADAPT it produced; on the "no findings" outcome it **is itself** the evidence referenced by the `source.adversarial-review-ref` field (§6.5.2).

The hook (§10.11.1 `adapt-applicability validation`), when BR / SR / SPEC are created without `source.adapt` , checks for an AR with status `issued` and a valid verdict for the corresponding TZ. The absence of such a record is fatal: creation is blocked until the adversarial review is passed.

7.4.1.3 Prohibition of silent skipping

Creating BR / SR / SPEC from a TZ **without an adversarial review** (skipping ADAPT without a verdict) is a violation of the standard. This preserves the Source-of-Truth inversion (§2.3): "no findings" is a **recorded assertion** by the adversarial reviewer, not a silent assumption by the Architect. The verdict MUST be issued as an AR (§7.4.6) with author and timestamp (V6) and be available on an auditor's request (§13.5).

When a delta-TZ is present, the same rule applies: a delta-ADAPT is created reactively, upon findings during the adversarial review of the delta-TZ (§7.6).

7.4.1.4 Multiplicity of ADAPTs per single TZ

Since the trigger is stage-agnostic (§7.4.1.1), a single unmodified TZ may have **zero or more** root ADAPTs (cardinality 0..N — a reformulation of MVR-3, §0.5):

- **zero** — adversarial review at all stages returned "no findings";
- **one** — a finding was discovered once (typically at import);
- **several** — findings rooted in the TZ arose at different derivation stages (import, then BR → SR decomposition, etc.).

Each ADAPT keeps a stable end-to-end `ADAPT-NNN` and records a `trigger-stage` field — the stage at which it was produced (§7.8.1). Multiplicity is the **regular** case, not an exception, and it does not weaken provenance: each BR / SR / SPEC still references exactly one ADAPT (through `source.adapt`) or the TZ directly (through `source.tz-section`) from which it is derived.

7.4.2 Immutability of the TZ

The TZ is a contractual document. After a TZ is registered in the substrate, its content **is not edited**. If during work it turns out that the TZ has an error / gap / contradiction, this is registered as a backward finding in ADAPT (§7.4.4), the client gives an answer, and the answer becomes part of ADAPT. The TZ remains immutable.

With a large number of edits, or when the scope changes, the client signs a delta-TZ as a new immutable document (§7.6).

7.4.3 Forward adaptation of the TZ

For each TZ section, the forward-interpretation section of ADAPT MUST contain:

Element	Obligatoriness	Purpose
Exact reference to TZ§N.N	REQUIRED	provenance
Quote from the TZ (or a paraphrase with an explicit marker)	REQUIRED	Context
Engineering interpretation of the section	REQUIRED	Translation of the client's language → the requirements language
Term mapping (client → engineer)	REQUIRED if applicable	Term disambiguation
Filled-in scenarios	REQUIRED if the TZ implies them	Explicit recording of implicit cases
Scope clarification (in / out)	REQUIRED	Scope

References from the forward interpretation to BR/SR/SPEC	auto-derived	Trace chain (§7.7)

7.4.4 Backward findings and their categories

The backward-findings section of ADAPT records discovered problems across seven normative categories. **The list of categories is closed at v1.0**; adding new categories is done through the formal change procedure of the standard (see chapter 13). The general closed-list policy and the master index — §1.7.5.

ID	Category	What is recorded
`contradiction`	Contradiction	Internal contradictions in the TZ (§A vs §B)
`gap`	Gap	The TZ is silent about something without which implementation is impossible
`hidden-assumption`	Hidden assumption	An engineer's assumption that may be wrong
`feasibility`	Feasibility	A technically infeasible or disproportionately expensive requirement
`regulatory`	Regulatory	A requirement that touches legislation / compliance
`terminology`	Terminology	An unclear TZ term with several possible meanings
`scope`	Scope	An unclear scope

Each backward-finding record has a **stable ID** (`B-NNN`), immutable after creation. The ID is not reused even for withdrawn records (audit log).

7.4.5 Lifecycle of a single backward-finding record

```
open → asked-to-client → answered → resolved → frozen
              ↑                          │
              └──────── revised ─────────┘   (if the answer needs clarification)
```

Status	What it means
`open`	The engineer recorded it; not yet put to the client
`asked-to-client`	The question is put to the client as part of an ACTZ (§7.13); signing is awaited; the date is recorded
`answered`	The client took a decision; the decision is recorded as a clause of a **signed** ACTZ
`resolved`	The engineer integrated the decision into the forward interpretation; the record carries `decided-in: ACTZ-NNN §M`
`revised`	The client's decision is vague; the question is put again in a new ACTZ. Transition back to `asked-to-client`

`frozen`	After ADAPT approval — changes are impossible

The link to an ACTZ is mandatory. Every record that requires a client decision MUST carry the field `decided-in: ACTZ-NNN §M` — a reference to the clause of a **signed** ACTZ in which the decision is recorded (§7.13). A reference to an unsigned ACTZ, and equally a dangling reference to a non-existent ACTZ, is **fatal** (§10.11.1).

ADAPT approval (§7.5) is prohibited while at least one backward-finding record is in status `open` / `asked-to-client` / `answered` / `revised`. All such records MUST be in `resolved` before approval — that is, each MUST carry a signed client decision.

The reverse direction MUST also be expressed: a decision taken by the client on their own initiative (an ACTZ with no parent finding, §7.13.2) MUST be **reflected in the interpretation** — ADAPT reacts to the ACTZ. A signed decision not reflected in any ADAPT is **fatal**: it is an obligation absent from the requirements.

7.4.6 AR — the adversarial-review record

AR (Adversarial-Review Record) is an evidence record, immutable once issued, that captures the fact and the outcome of the mandatory adversarial review (§7.4.1.2).

AR belongs to the class of **evidence records**, not to the requirements graph: it has no parents and no children, it is not decomposed, it is not verified by test cases, and it is neither a requirement nor a specification. AR therefore **does not extend** any closed list of the standard (§1.7.5): it gives a definite target to the already existing `source.adversarial-review-ref` field rather than introducing a new type of requirements artifact.

7.4.6.1 Identity and multiplicity

The identifier is `AR-NNN`, sequential within the parent TZ and immutable once created (mirroring `ADAPT-NNN`, §7.4.1.4). The year is carried by the parent TZ (`TZ-YYYY-NNN`); for a delta-TZ, numbering is scoped to that delta-TZ (§7.11).

Cardinality is **zero or more** ARs per TZ, exactly as for ADAPT. The review at TZ import is always mandatory and yields the first AR; at derivation stages an AR is issued whenever a review is actually conducted. The unit of granularity is the **review event**: several artifacts derived from one review event without new questions reference **one** AR.

Completeness is ensured not by mandating "an AR at every node" but by the node-provenance invariant:

> *Every BR / SR / SPEC references exactly one source of adaptation — `source.adapt` **or** `source.adversarial-review-ref`. The adversarial review is mandatory at TZ import; at derivation stages its outcome is recorded as an AR whenever the review is conducted.*

7.4.6.2 Mandatory fields

Field	Obligation	Meaning
`id`	REQUIRED	`AR-NNN`; immutable
`tz-ref`	REQUIRED	The (delta-)TZ under review

`trigger-stage`	REQUIRED	Review stage: `import-tz` / `decompose-br` / `decompose-sr` / `spec` / `tc` (the list is aligned with ADAPT's `trigger-stage`, §7.8.1)
`reviewer.vendor` + `reviewer.model`	REQUIRED	The adversarial reviewer's model; it MUST differ from the primary agent's model (§7.10.2)
`verdict`	REQUIRED	`findings-present` or `no-findings`
`produces-adapt[]`	Conditional	REQUIRED and non-empty when `verdict: findings-present`; absent or empty when `no-findings`
`status`	REQUIRED	`draft` / `issued` / `superseded`
`superseded-by`	Conditional	REQUIRED when `status: superseded`
`signature.author` + `signature.timestamp`	REQUIRED for `issued`	Substrate capability V6

On the `findings-present` outcome the AR body carries only pointers to TZ sections and to the `B-NNN` records it produced; **the findings themselves live in ADAPT and are not duplicated in the AR**. On the `no-findings` outcome the body carries a short justification of the conversion's unambiguity.

7.4.6.3 Lifecycle

```
draft → issued → superseded
```

Status	Meaning
`draft`	The review is conducted, the verdict is being drawn up; referencing this AR from `source.adversarial-review-ref` is **prohibited**
`issued`	The verdict is issued and signed by the reviewer (V6); the record is **immutable**
`superseded`	A later review at the same stage displaced the earlier verdict; a terminal state, the record is retained for audit (V1)

An issued AR is not edited. Its only exit is `superseded`, when a repeat review at the same stage issues a new verdict: a new AR is then issued and the previous one receives `status: superseded` and `superseded-by`. The supersession model is the one already defined for ADAPT (§7.6.4); no separate quality gate is introduced for AR (§10.11.1).

A `source.adversarial-review-ref` pointing at a non-existent AR, or at an AR in status `draft` or `superseded`, is **fatal**: the gate blocks the change (§10.11.1).

7.5 ADAPT approval — the Architect's signature

ADAPT is an **internal artifact of interpretation**. Its audience is the engineer, the AI agent, the adversarial reviewer, and the verifier; the client neither sees nor signs it. Obligations towards the client live in ACTZ (§7.13).

ADAPT moves from status `answered` to `approved` after the **Architect's signature** (an atomic change unit, substrate capabilities V2 + V3 from §3.3):

Signature	Who	What it confirms
Architect signature	The Architect on the implementer's side	All backward findings are handled (each is in status `resolved` and carries `decided-in` pointing at a clause of a **signed** ACTZ, §7.4.5); the forward interpretation is technically feasible and consistent with the decisions taken

The substrate-native implementation of the signature is a combination of V3 (diff & review) + V6 (author + timestamp). The concrete mechanism (digital signature / approval process / dual review attestation) is chosen by the implementation and recorded in the conformance manifest (§3.7).

Why there is no client signature here. A client signature on ADAPT was a fiction: the client signed an engineering document they did not read and could not assess. The boundary is drawn **by audience**: everything shown to the client and approved is an obligation (ACTZ, dual signature); everything not shown is an interpretation (ADAPT, the Architect's signature). The content of a record is thereby not up for debate: there is no longer any need to argue whether something "is an interpretation or already a change" — it suffices to ask whether the decision was put to the client.

After approval, ADAPT is **immutable** on par with the TZ. Further changes are made only by adding a new artifact with a typed link, through one of three non-overlapping mechanisms: delta-ADAPT for a delta-TZ (§7.6.1), errata for an interpretation error (§7.6.3), or supersession of a previously correct decision (§7.6.4). The frozen ADAPT itself is not edited in any of them.

An erratum that touches no decision of a signed ACTZ (a fix to an interpretation error that does not affect the obligations towards the client) **does not concern** the client and is signed by the Architect alone (§7.6.3).

7.6 Delta-TZ and delta-ADAPT

7.6.1 Workflow

A delta-ADAPT follows the same reactive rule of §7.4.1 as the root ADAPT: it is created upon findings during the adversarial review of the delta-TZ.

When a delta-TZ arrives from the client:

1. The client registers the delta-TZ in the substrate as a new immutable document (`TZ-YYYY-NNN-delta-N`).
2. The client signs the delta-TZ (V6 author + timestamp).
3. The adversarial reviewer (§7.10.2) conducts a review of the delta-TZ and issues a verdict.
4. **If the verdict is "findings present"** — the Architect / AI agent creates a delta-ADAPT (`ADAPT-NNN-delta-N`) with the frontmatter field `parent-adapt: ADAPT-NNN` and `source-tz: TZ-YYYY-NNN-delta-N` . The Forward interpretation covers only the delta-TZ sections. Backward findings are recorded against the delta-TZ. Approval — the Architect's signature of §7.5; the client's decisions on delta-TZ findings are recorded in signed ACTZ protocols (§7.13).

5. **If the verdict is "no findings, no clarifications"** — no delta-ADAPT is created. The verdict is recorded in the substrate with V6 author + timestamp. BR / SR / SPEC changed as a result of the delta-TZ reference `source.tz-section: TZ-YYYY-NNN-delta-N` directly.

In both cases the evidence of the adversarial review (the verdict) MUST be machine-accessible for audit (hook §10.11.1 reference-validation).

7.6.1bis Example: a trivial delta-TZ

Delta-TZ: "rename the 'username' field on the registration form to 'email'".

1. The client signs `TZ-YYYY-NNN-delta-1`.
2. The adversarial reviewer checks: the term is unambiguous (`email` is a standard engineering term), the scope is clear (one field on one form), there are no questions for the client.
3. Verdict: "no findings, no clarifications"; recorded in the substrate.
4. The delta-ADAPT **is not created**.
5. SR-NN with the behavior "POST /auth/sign-up accepts email" receives an update: `source.tz-section: TZ-YYYY-NNN-delta-1 §1`. The SR `parent` BR-NN does not change.

7.6.2 Delta-ADAPT chain

```
ADAPT-001 (from TZ-YYYY-NNN main)
  └─ ADAPT-001-delta-1 (from TZ-YYYY-NNN-delta-1)
       └─ ADAPT-001-delta-2 (from TZ-YYYY-NNN-delta-2)
            └─ ADAPT-001-delta-3 (from TZ-YYYY-NNN-delta-3)
```

The chain is strictly sequential: applying delta-ADAPTs MUST proceed in order (see the cross-substrate version pin V5 in §3.3.5). Renumbering or reordering delta-ADAPTs in the chain is prohibited.

7.6.3 Errata for an already approved ADAPT

If, a considerable time after approval, it is discovered that the forward interpretation of the TZ in ADAPT-NNN is wrong, or the resolution of a backward finding is recorded incorrectly — two permissible outcomes:

Outcome	Artifact
The TZ contains an ambiguity discovered late	delta-ADAPT with a new backward-finding record and a client answer
Wrong interpretation (an engineer's error)	errata-ADAPT-NNN-M as a separate artifact. If the fix touches a decision of a signed ACTZ (changes an obligation), a **new ACTZ** with a dual signature is REQUIRED; if it does not, the Architect's signature suffices

In both outcomes the **frozen ADAPT is not edited**. Only the addition of new artifacts with an explicit typed link.

7.6.4 Supersession of an approved ADAPT

Delta and errata do not cover one case: a previously accepted decision was **correct**, but requirements formed later **contradict** it under a new understanding. This is not an engineer's error (errata) and not a new contract from the client (delta) — it is the cancellation of a previously correct decision. For it, a third mechanism is introduced — **supersession** (`supersession`).

Mechanism	When	Nature
delta-ADAPT (§7.6.1)	a delta-TZ arrived from the client	new source: the contract changed
errata-ADAPT (§7.6.3)	the former interpretation was **erroneous**	a correction of what was always wrong
superseding ADAPT (§7.6.4)	the former decision was **correct**, but requirements formed later contradict it	cancellation of a previously correct decision under a new understanding

Supersession rules:

1. **Superseding artifact.** A new `ADAPT-NNN` is created with the frontmatter field `supersedes: ADAPT-MMM` and a mandatory `supersession-rationale` referencing the concrete contradicting requirement (`BR` / `SR` / `SPEC` ID) and its source. The superseded ADAPT receives an automatically derived back-reference `superseded-by: ADAPT-NNN` (§7.8.1).
2. **Signature.** If the superseded decision had a **contractual outcome** (it was recorded as a clause of a signed ACTZ) — supersession REQUIRES a **new ACTZ with a dual signature**: a decision agreed with the client cannot be cancelled unilaterally, and the cancelling protocol sets `superseded-by` on the earlier one (§7.13.4). If the fix touches no decision of a signed ACTZ, the Architect's signature suffices (by analogy with the errata rule, §7.6.3). No separate QG is introduced — supersession passes through the same QG-3 (§7.5, §10.8.5).
3. **State.** The superseded `ADAPT-MMM` moves to the dedicated terminal state **`superseded`** — separate from `obsolete` (becoming outdated) and from `frozen` . `superseded` is immutable and is **retained** for audit (immutable history, substrate capability V1); it is not deleted. The transition is regulated in §10.8.5.
4. **Redirection of derivatives.** All `BR` / `SR` / `SPEC` with `source.adapt: ADAPT-MMM` MUST be either redirected to the superseding ADAPT or re-derived. A dangling `source.adapt` reference to an ADAPT in status `superseded` is **fatal**: the gate `check-adapt-supersession.js` blocks it (§10.11.1), by analogy with reference-validation.
5. **Additivity.** Like delta and errata — the superseded ADAPT **is not edited**; only a new artifact and the typed link `supersedes` / `superseded-by` are introduced.

Supersession is an extending capability: a single ADAPT without supersessions remains a valid special case, and migration of existing projects is not required.

7.7 Relationship of ADAPT to other artifacts

7.7.1 BR / SR / SPEC reference ADAPT through `source.adapt`

```yaml
# Frontmatter SR (example)
source:
  adapt: ADAPT-001
  adapt-section: "Forward §3"   # forward interpretation; canonical section
identifier — Forward §*
  tz-section: "§3.4"            # for traceability; the primary source remains
the TZ
```

The `source.adapt` field is conditional: present when the TZ → RENAR conversion required ADAPT; omitted when the adversarial reviewer returned a "no findings" verdict — then the `source.adversarial-review-ref` field is REQUIRED (§7.4.1). The `source.tz-section` field is REQUIRED always (the dual trace chain).

7.7.2 The full trace chain

```
TC-NN  →  verifies SR-12  →  derived from ADAPT-001 §4 (forward interpretation)
                                  │     │
                                  │     └─ resolves B-007 (was: contradiction,
                                  │            answered by client 2026-03-15)
                                  │
                                  └─ interprets TZ-YYYY-NNN §3.4
```

When verifying from a test case, one can reach: (a) the source TZ section, (b) the forward interpretation of that section, (c) the implementer's questions on backward findings, (d) the client's answers, (e) the derived BR / SR / SPEC.

7.7.3 TR does not reference ADAPT directly

A TR (task) references SR / SPEC, and those already reference ADAPT. The task implementer **does not access ADAPT directly** — all necessary interpretations are already contained in SR / SPEC. If the implementer discovers an ambiguity in an SR, the root determines the outcome (§7.4.1.1):

- **root in decomposition** (not in the TZ — for example, an imprecise SR wording, a missed `constrained-by[]` link) — resolved by clarifying the SR / SPEC **without** ADAPT;
- **root in the language or intent of the TZ** — a backward finding is registered in ADAPT. If an ADAPT for this TZ already exists, a new record is added; if an ADAPT has not yet been created (the verdict at import was "no findings"), a **new** ADAPT is created at the current stage (§7.4.1.1). This is the regular case, not an exceptional one.

7.8 ADAPT schema

7.8.1 frontmatter (mandatory fields)

```yaml
---
id: ADAPT-NNN                          # immutable; NNN sequential per project
title: "TZ adaptation <name>"
type: ADAPT
trigger-stage: import-tz               # stage that produced the ADAPT (§7.4.1.4):
                                       # import-tz | decompose-br | decompose-sr |
spec | tc

source-tz:
  id: TZ-YYYY-NNN
  signed-date: "<ISO-date>"
  signed-by-client: "<name + role>"
  document-version-ref: "<substrate-native version identifier>"   # V5 pin (see
§3.3.5)

parent-adapt:                          # for delta-ADAPT
  id: ADAPT-NNN
  delta-tz: TZ-YYYY-NNN-delta-N

supersedes: ADAPT-MMM                  # only for a superseding ADAPT (§7.6.4);
omitted otherwise
superseded-by: ADAPT-NNN               # auto-derived; set on the superseded ADAPT
supersession-rationale: >              # mandatory if supersedes: is present
  "<reference to the contradicting BR/SR/SPEC ID + its source; rationale for
cancellation>"

status: draft | review | asked | answered | approved | frozen | superseded
created: "<ISO-date>"
last-updated: "<ISO-date>"

approval:                               # mandatory for approved
  # client-signature IS ABSENT: an ADAPT is an internal interpretation (§7.5).
  # The client's obligations are carried by an ACTZ (§7.13) with the dual
signature.
  architect-signature:                  # mandatory for approved
    signed-by: "<name>"
    role: architect
    signed-at: "<ISO-datetime>"

generates-requirements: []             # auto-derived; BR/SR from this ADAPT
generates-specs: []                    # auto-derived; SPEC-* from this ADAPT
open-questions-count: integer          # auto-derived; mandatory 0 for approved
resolved-questions-count: integer

ai-provenance:                         # mandatory if the ADAPT draft was AI-
generated
  generated-by: "<vendor>-<model>@<date>"
  prompt-template: "<template-path>@<version>"
  context-tokens: integer
  output-tokens: integer
  human-edits: boolean                 # mandatory true for approved — the
```

```
Architect reviewed the text (§7.5)
---
```

Note: ADAPT exists in only one mode (the Architect's signature, the full lifecycle of §7.4.5). Reactivity (§7.4.1) is expressed **at the level of creation**: if the adversarial reviewer returned a "no findings, no clarifications" verdict, ADAPT is not created at all, rather than being created in a simplified form.

7.8.2 Body structure (mandatory sections)

The mandatory sections of the ADAPT body:

1. **Summary** — 3–5 paragraphs for one-page reading by the Architect and the adversarial reviewer (§7.5).
2. **Term mapping** — a "client → engineer" table.
3. **Forward interpretation (the Forward section)** — a section for each TZ section with the mandatory elements from §7.4.3.
4. **Backward findings** — all records with the lifecycle from §7.4.5.
5. **Backward-findings summary** — a statistical table by categories and statuses.
6. **Derived-artifacts table** — an auto-derived list of BR / SR / SPEC.
7. **ADAPT change history** — substrate-native, auto-generated.

Optional sections are at the implementation's discretion and are not regulated.

7.9 Quality Gates for ADAPT

ADAPT has a dedicated state machine. Details in chapter 10 §10.8. Brief summary:

Gate	Precondition	Postcondition
QG-ADAPT-draft	ADAPT created; frontmatter present	The forward interpretation covers all TZ sections
QG-ADAPT-review	Forward interpretation filled in; initial backward findings in `open`	All backward findings in `open` or `asked-to-client`
QG-ADAPT-asked	All backward findings in `asked-to-client`; the questions have been put to the client as part of an ACTZ (`status: sent`)	Awaiting the protocol's signing
QG-ADAPT-answered	All backward findings in `answered`; resolution begun	Ready for finalization
QG-ADAPT-approve	All backward findings in `resolved` and carry a `decided-in` pointing to a signed ACTZ; the Architect's signature is ready	ADAPT immutable; generation of BR / SR / SPEC is permitted
QG-ADAPT-frozen	`approved`	Further changes — only through delta-ADAPT or errata

The substrate hooks (§3.3.3 V3, §3.3.5 V5) MUST:

- Block the transition to `approved` when `open-questions-count > 0`.

- Block the creation of BR / SR / SPEC with `source.adapt` on an ADAPT in a status below `approved`.
- Recompute `open-questions-count` / `resolved-questions-count` after each change.

7.10 ADAPT and AI generation

7.10.1 The AI agent creates a draft ADAPT

On a "findings present" verdict from the adversarial reviewer (§7.4.1), the AI agent creates a **draft ADAPT**: the forward interpretation by section, an attempt to detect contradictions / gaps / terminology ambiguities, a first version of the term mapping. This draft is a starting point for the Architect, not the final artifact.

7.10.2 The adversarial reviewer

Application of the **adversarial review principle** (a separate AI agent-critic with a **different model**; the procedure — guide/07 §3.5): it looks for what the primary agent missed — missing backward findings. If the adversarial reviewer finds at least three serious new findings, the primary draft is returned for rework.

7.10.3 The client does not communicate with the AI directly

Questions on backward findings are aggregated by the Architect into a human format **before being sent to the client**. The Architect MAY remove duplicates, rephrase into the client's language, merge related ones. The client sees a prepared list of questions, not the raw output of the AI agent. The client's answer (in any form: text, video call, letter) is transcribed by the Architect into ADAPT with an indication of the channel and authentication.

7.11 Storage scheme

ADAPT documents are stored in the `adapt/` subfolder of the requirements substrate:

```
[project]/
  adapt/
    ADAPT-001-main.md
    ADAPT-001-delta-1.md
    ADAPT-001-delta-2.md
    ADAPT-002-main.md
    errata/
      errata-ADAPT-001-1.md
  ar/
    AR-001.md
    AR-002.md
  tz/
```

```
TZ-YYYY-NNN.md
TZ-YYYY-NNN-delta-1.md
```

Adversarial-review records (§7.4.6) are stored in the `ar/` subfolder. They exist even when no ADAPT was created (the `no-findings` verdict) — hence the subfolder is independent of `adapt/`.

The concrete file structure on a concrete substrate is substrate-specific (see guide/03 for distributed VCS; guide/04 for a document-oriented store).

7.12 The relationship of the TZ and the RENAR description

7.12.1 Two languages with a variable distance

The TZ and the RENAR description are two different artifacts with different natures:

Parameter	TZ	RENAR description
Target reader	The client (human)	The AI agent (§0.2.1) and the human verifier
Language	The client's language (business domain, contractual vocabulary)	The requirements language (canonical IDs, closed lists, formal frontmatter and graph links)
Completeness	Allows defaults, incompleteness, ambiguity of wording	Allows no defaults; completeness is the lower bound of machine-enforceability (§0.3)
Evolution	Incremental: the first TZ describes the system completely, the subsequent ones — only the delta (§7.6)	Non-incremental: before changing the system, a new full version of the RENAR description is created, and only then does the AI agent make changes in the implementation
Status after signing	An immutable contractual document (§7.4.2)	Evolves through an approved delta-ADAPT or (when no ADAPT is created, §7.4.1) through source.tz-section directly

The distance between the two languages is **variable**. Sometimes TZ sections are worded unambiguously enough for the AI agent to convert them into BR/SR/SPEC without loss of meaning. Sometimes the reverse: the TZ contains a contractual phrase that has several engineering interpretations, or omits behavior without which implementation is impossible.

7.12.2 ADAPT — a reactive bridge between languages

ADAPT exists only when a gap arises between the languages. Its forward interpretation (forward, §7.4.3) is a **translation** of concrete TZ sections from the client's language into the requirements language. The backward findings (backward, §7.4.4) are a formalization of the fact that, during translation, gaps, ambiguities, or contradictions were discovered that require agreement with the client.

Three consequences follow from this nature:

1. **ADAPT is a reactive artifact by design.** The existence of a gap between the languages is a necessary condition for creation (§7.4.1.1); a formal ADAPT without content loses its meaning for audit and devalues the other ADAPTs.

2. **The adversarial reviewer is the only one who declares the absence of a gap.** "ADAPT is not needed" is a recorded verdict by a different model (§7.10.2, §7.4.1.3), not a silent assumption by the Architect.
3. **The form of ADAPT is the only one.** It is created in the full form (the Architect's signature §7.5, all body sections §7.8.2, the full lifecycle §7.4.5); intermediate "light" forms do not exist.

7.12.3 Why the RENAR description is always complete

The RENAR description is non-incremental by design: the AI agent (§0.2.1) is unable to reliably "fill in" changes relying only on the delta. A full new version of the RENAR description is the only form that guarantees that:

- machine-enforceable invariants (completeness, graph consistency, lifecycle states) are applicable to the description **as a whole**, not to the diff;
- the adversarial reviewer works on the full artifact, not on the delta;
- the subsequent steps (decomposition, TC generation, implementation — `guide/00-quickstart §"The two regular scenarios"`) are performed on the up-to-date full picture of the system.

A delta-TZ is incremental at the **source** level (the contractual side); the full new version of the RENAR description is assembled by the AI agent from the parent RENAR + either the delta-ADAPT (if it was created) or directly accounting for the delta-TZ through `source.tz-section`.

7.12.4 Relationship to the Source-of-Truth inversion

The TZ is a contractual artifact but **not** the Source of Truth about system behavior (§2.3). The Source of Truth is the RENAR description (BR / SR / SPEC / TR / TC). The TZ is the source from which the RENAR description is derived **either** through ADAPT (when there is a gap) **or** directly through `source.tz-section` (when there is no gap); code is a derived artifact of the implementation of the RENAR description. ADAPT and §7.12 fix the dual inversion: the contractual source (TZ) → the Source of Truth (RENAR description) → code. ADAPT is embedded in the chain reactively, when the bridge between the languages is needed.

7.13 ACTZ — the TZ Clarification Protocol

7.13.1 Purpose and the boundary with ADAPT

ACTZ (`ACTZ-NNN`, Agreed Clarification of TZ; in contractual usage — "**TZ Clarification Protocol No. N**") is an artifact of the **contractual circuit**: a batch of questions, proposals, and decisions put to the client and signed by both parties.

The boundary between ACTZ and ADAPT is drawn **by audience**, not by the content of a record:

> ***Everything shown to the client and approved is an obligation. Everything not shown is an interpretation.***

	ADAPT — interpretation	**ACTZ** — approval

Content	Translation of the TZ language into the engineering one, term mapping, filled-in scenarios, backward findings, forward interpretation	Questions, proposals, and decisions put to the client. Worded in the language of obligations ("the button is named X"), not of interpretation
Audience	Engineer, AI agent, adversarial reviewer, verifier	The client and the parties to the contract
Signature	The Architect's only (§7.5)	**Dual**: client + implementer
Weight	Internal circuit	**Contractual**

The criterion is binary and checkable: the fact "put to the client and signed" either holds or does not. The argument over the materiality of an edit ("is this a clarification or already a change?") disappears by construction.

7.13.2 Origin and multiplicity

An ACTZ arises in two ways, both regular:

1. **From a backward finding.** A finding that requires a client decision (§7.4.4) is put into an ACTZ; once signed, the record receives `decided-in: ACTZ-NNN §M` (§7.4.5).
2. **On the client's initiative, with no finding.** The client proposes a decision on their own ("let us rename the button"). Such an ACTZ references only the TZ; it has no parent finding. Once signed, the decision MUST be reflected in ADAPT — the interpretation reacts to the protocol.

Cardinality:

Relation	Cardinality	Explanation
TZ : ACTZ	**1 : 0..N**	Zero protocols is lawful: a conversion with no questions and no client initiatives produces no documents
ADAPT : ACTZ	**1 : 1..N**	The findings of a single ADAPT are closed by several protocols across rounds of clarification

A late ACTZ is the regular case, not an exception. A protocol may arise at **any stage in the life of the engagement**, including from triaging the customer's remarks after a demonstration. The rule is stage-agnostic — symmetrically to the reactive ADAPT (§7.4.1).

7.13.3 Mandatory fields

Field	Obligation	Meaning
`id`	REQUIRED	`ACTZ-NNN`; sequential within the parent TZ; immutable
`tz-ref`	REQUIRED	The TZ the protocol relates to
`resolves[]`	Conditional	The `B-NNN` records in ADAPT that the protocol closes; absent for an ACTZ raised on the client's initiative
`decisions[]`	REQUIRED	Decision clauses; each is worded in the language of obligations and carries a stable number `§M`

`annexes[]`	Conditional	New versions of TZ annexes approved by this protocol (§7.13.5)
`status`	REQUIRED	`draft` → `sent` → `signed` → `superseded`
`client-signature`	REQUIRED for `signed`	The client or a client representative with authority (V6: author + timestamp)
`vendor-signature`	REQUIRED for `signed`	The implementer (V6)
`superseded-by`	Conditional	REQUIRED when `status: superseded`

7.13.4 Lifecycle

```
draft → sent → signed → superseded
```

Status	What it means
`draft`	The protocol is being prepared; not yet put to the client. Referencing it from `decided-in` is **prohibited**
`sent`	Put to the client, signing is awaited
`signed`	Signed by both parties; **immutable**, carries contractual weight
`superseded`	A later signed ACTZ cancelled the decision; a terminal state, the record is retained for audit (V1)

A signed ACTZ **is not edited**. A correction is made only by a new ACTZ cancelling the earlier decision (`superseded-by`); the supersession model is the one already defined for ADAPT (§7.6.4).

What is put to the client is a **prepared list of decisions**, not the raw output of the AI agent (§7.10.3): the Architect aggregates and rewords the questions.

7.13.5 TZ annexes

Annexes (mapping tables, reference lists, mockups) are an **integral part of the TZ**; they are not a separately addressable entity. A clarification to an annex is formalized as a **new version of the annex attached to an ACTZ** and approved by the same dual signature (`annexes[]`).

This closes the case where the client's decision is not a text but a new version of a table: the former version remains immutable (V1), and the new one enters the effective TZ through the protocol that signed it.

7.14 The effective TZ — the acceptance benchmark

The **effective TZ** is the benchmark against which the system is delivered and accepted:

> ***Effective TZ = the initial TZ (annexes included) + all signed ACTZ protocols.*** *On a discrepancy between documents, the* ***later signed*** *document prevails.*

Acceptance tests AT (§9.19) are derived against the effective TZ; the acceptance report is built from it as well.

ADAPT is not part of the acceptance benchmark. The internal interpretation is taken out of the contractual circuit entirely — the client answers only for what they signed and understood. This is the very point of the split: acceptance becomes legally clean.

QG-4 (§10.4.2) is a separate **optional** gate on the business result (`achievement ≥ 80%`). It **does not replace** acceptance against the effective TZ and is not conflated with it: a business goal MAY be met while the contract is not conformed to, and vice versa.

7.15 Relationship to other chapters

Chapter	Relationship
02 Methodology positioning	ADAPT is a consequence of Statement 2 (two-way adaptation instead of "throwing the specification over the wall")
06 Requirements hierarchy	BR / SR reference ADAPT through `source.adapt`
08 Specifications	SPEC-* also reference ADAPT through `source.adapt`
09 Test cases	TC, through SR / SPEC, returns to ADAPT for the full trace chain
10 Lifecycle and QG	the ADAPT state machine + QG-ADAPT-* gates
03 Substrate versioning	The Architect's signature on an ADAPT and the ACTZ dual signature (V6) + atomic approval (V2 + V3); immutability after approved (V1); delta-ADAPT through V4
13 Conformance	The adversarial review of each TZ is a mandatory clause; ADAPT is created reactively (§7.4.1)

08. Specifications — 11 SPEC types

> ***Part of the RENAR Standard v1.0*** · *← Table of contents*

8.1 Why a separate specification axis

Take the requirement "the system creates an order." It says **what** must happen — but it is silent about **how** the system is built to do it: what its API contract is, in which table the order lives, under which access rules, on which screen. Cramming all of that into the requirement itself does not work — it turns into mush. So RENAR splits the description into two axes: **behavior** (BR / SR / TR, chapter 6) and **structure** — specifications, SPEC.

A specification is not a "more detailed SR" and not its child. A single "create order" requirement typically rests on five to seven specifications at once (architecture, API, data, process, security, screen), so the link between the axes is a graph of typed edges (`constrained-by[]` , `implements-spec[]`), not a tree. There are exactly eleven specification types, and the list is closed: `SPEC-ARCH` , `API` , `DATA` , `INT` , `PROC` , `UI` , `AI` , `SEC` , `OPS` , `TEST` , `DOC` — a new type is introduced only through the formal change procedure of the standard (chapter 13).

A specification describes **three different subjects**, and they MUST NOT be conflated. Nine types describe **how the system is built**. `SPEC-TEST` describes **the construction of the proof** of its conformance: the test bench of an integration system is not an installation but an engineering product (emulators, datasets on both sides of every integration, rules for reconciling results against the data of foreign systems). `SPEC-DOC` describes **the composition of the delivered result**: the documentation the client receives as part of the delivery.

8.2 Architectural decision: SPEC is a parallel axis, not children of SR

8.2.1 Two axes of describing the system

Requirements and specifications answer different questions:

Axis	Artifacts	Question
Behavioral	BR / SR / TR (chapter 6)	What the system must do
Structural	SPEC-* (9 types)	How the system is structurally built to fulfill those requirements
Evidential	`SPEC-TEST`	On what bench and on what data conformance is proven (§8.3.2)
Delivery	`SPEC-DOC`	What enters the delivery to the client besides the system itself

8.2.2 SPEC as a parallel axis: links through a typed graph

The links between the requirements axis (BR / SR / TR) and the SPEC axis are organized as a **dependency graph**, not a tree of parents. An SR has exactly one parent in the requirements tree (BR), but many typed

`constrained-by[]` edges to SPEC. A single SR MUST reference every SPEC that constrains its behavior on the API / data / UI / process / security / ops axes; conversely, one SPEC MAY constrain many SR.

Example: the SR "create order" rests on SPEC-ARCH (where the orders component lives), SPEC-API (the endpoint contract), SPEC-DATA (the table schema), SPEC-PROC (workflow), SPEC-SEC (access rules), SPEC-UI (the form).

Normatively: every `SR.constrained-by[]` and `TR.implements-spec[]` edge MUST reference one of the closed SPEC categories listed in §8.3; ad-hoc categories are not allowed (see §1.7 closed-list policy).

```
Requirements tree (behavioral axis):       Parallel specification axis:

BR                                          SPEC-ARCH    SPEC-API
 └── SR  ←──── constrained-by[] ────►       SPEC-DATA    SPEC-INT
      └── TR ─── implements-spec[] ──►      SPEC-PROC    SPEC-UI
                                            SPEC-AI      SPEC-SEC
                                            SPEC-OPS     SPEC-TEST
                                            SPEC-DOC


Requirements tree:
  SR.parent              → BR              (single parent)
  TR.parent              → SR              (single parent)

Link graph (typed edges):
  SR.constrained-by[]    → SPEC-*
  TR.implements-spec[]   → SPEC-*
  SPEC-*.depends-on[]    → SPEC-*          (between specifications)
  SPEC-*.referenced-by[] → SR / TR          (auto-derived inverse)
```

8.2.3 Rationale

Argument	Consequence
SPEC and SR answer different questions	SPEC does not refine an SR at a deeper level — it is a separate category of description
One SR rests on 5–7 SPEC	A "SPEC as parent of SR" tree leads to multiple parenthood
Industry standards (arc42, C4, OpenAPI, BPMN, ERD) live in parallel with requirements	RENAR follows this proven practice
An AI agent can parallelize SR and SPEC generation	Without one type blocking the other

8.3 The closed list of eleven SPEC types

Type	Purpose	Industry reference

`SPEC-ARCH`	System / subsystem architecture: contexts, containers, components, deployment view, quality attributes	arc42, C4 model (Brown), ISO/IEC/IEEE 42010
`SPEC-API`	API contracts: REST / GraphQL / gRPC / async events; versioning, error model, rate limits	OpenAPI 3.x, AsyncAPI 2.x, gRPC IDL
`SPEC-DATA`	Data model: schema, ERD, indices, migrations, retention, PII classification	ISO/IEC 11179, JSON Schema
`SPEC-INT`	Integration: interaction between subsystems and external systems; protocols, contracts, SLA	Enterprise Integration Patterns (Hohpe)
`SPEC-PROC`	Process / workflow: business processes, state machines, saga, choreography, orchestration	BPMN 2.0, ISO/IEC 19510
`SPEC-UI`	UI / UX: screens, navigation, user journeys, accessibility, i18n, baseline images	Material Design / Apple HIG, WCAG 2.2
`SPEC-AI`	AI / ML: model cards, RAG, prompt engineering, eval strategy, cost budget	ISO/IEC 23894, NIST AI RMF
`SPEC-SEC`	Security: authn / authz, threat model, secrets management, data classification	STRIDE, OWASP ASVS, ISO/IEC 27001
`SPEC-OPS`	Operations: deployment, observability, SLO / SLA, runbook, disaster recovery	Google SRE, ITIL v4, ISO/IEC 20000
`SPEC-TEST`	Test benches and data: topology, emulators and sandbox instances of external systems, run configuration (what is mocked, what is live), datasets on both sides of every integration, rules for reconciling results against the data of foreign systems, volume and anonymization	ISO/IEC/IEEE 29119-4 (test techniques), Testcontainers, Pact
`SPEC-DOC`	Delivered project documentation: composition (user manual, administrator manual, training materials), the structure of each document (mandatory sections), the checkable requirements on it (completeness across roles and scenarios, version currency, format)	ISO/IEC/IEEE 26511, ISO/IEC/IEEE 26514

8.3.1 What did NOT make it into v1.0 (with rationale)

Candidate	Decision	Rationale
`SPEC-EVENT`	Not a separate type	Events / queues — part of SPEC-API (asynchronous APIs)
`SPEC-CONFIG`	Not a separate type	Feature flags / env vars / secrets — part of SPEC-OPS
`SPEC-PERF`	Not a separate type	Performance / NFR — part of SPEC-ARCH (quality attributes) or SPEC-OPS (SLO)
~~`SPEC-TEST-ENV`~~	**Decision reversed in v1.0**	The former decision ("test environments — part of SPEC-OPS") is reversed: see §8.3.2. The test bench is introduced as a separate type, `SPEC-TEST`

`SPEC-DOMAIN`	Not a separate type	Domain model — absorbed into SPEC-ARCH (decomposition) + SPEC-DATA (entities)
`SPEC-MIGRATION`	Not a separate type	Migration — part of SPEC-DATA (lifecycle)
`SPEC-COMPLIANCE`	Not a separate type	Compliance — links between SR/SPEC and regulations through `compliance-refs[]`, not a separate artifact

A reversed decision is kept in the table struck through rather than deleted: the history of decisions is part of the standard, and the reader is entitled to see what was revised and why.

8.3.2 Revision of the test-bench decision

The former decision identified the test bench with a **deployment environment** and sent it to `SPEC-OPS` (the `environments[]` field: dev / staging / prod). For a simple system that is correct: the bench there is staging plus a dataset. The decision is reversed for three reasons.

1. The bench of an integration-heavy system is a different object. It is not an installation of the system but **the construction of the proof**: emulators and sandbox instances of external systems; agreed datasets **on both sides** of every integration; rules for reconciling results against the data of foreign systems — the expected result lives **not in our system**; the run configuration (what is mocked, what is live). The standard already sensed this: §9.6.3 requires, for SPEC-INT, a run "against a real or sandbox counterparty — a mocked contract is not enough," yet it gave no carrier for describing that counterparty.

2. False invalidations — the decisive argument. §10.5.4 invalidates `verified` on **any** version increment of an artifact, automatically and without inspecting the cause. If a TC referenced the bench through SPEC-OPS, then an edit to the **deployment procedure** — unrelated to the bench — would drop **every** TC referencing that SPEC-OPS back into `approved`. Deployment is edited more often than the bench, and the body of evidence would be devalued regularly and falsely. A separate `SPEC-TEST` makes invalidation **precise**: TC are invalidated if and only if what they are valid against has changed.

3. An alien signature. Sensitive test data (volume, anonymization, the use of the client's real data) is agreed by the **client**. A client signature on a section of an operations document is alien; on a separate artifact it is natural.

The boundaries of the new types:

Type	Answers the question
`SPEC-OPS`	**How the system lives** in operation (deployment, observability, SLO, runbook)
`SPEC-TEST`	**How its conformance is proven** (benches, data, run configuration)
`SPEC-DOC`	**What enters the delivery** to the client (documents as a subject of the contract)

The SPEC-DOC ↔ SPEC-OPS boundary. The criterion is **inclusion in the composition of the delivered result**, and that composition is set by the contract **before** acceptance rather than settled during it:

- **The administrator manual is always `SPEC-DOC`**: if the client operates the system, the manual for the client's administrators enters the composition of the delivery. No "internal version" of this genre exists.

- **The runbook is always** `SPEC-OPS` : it is the internal operational regulation of the contractor's operating team; it is not presented to the client and is not a subject of acceptance.

Dual membership of a document is excluded by construction — the `TR.implements-spec[]` edge is always unambiguous.

Closed-list policy: if subsequent work reveals that one of the excluded types is genuinely needed — it is added through the formal change procedure of the standard with rationale.

8.4 Common schema (shared frontmatter fields)

All 11 SPEC types share a common set of frontmatter fields. Type-specific fields are added as extensions on top (§8.5). The full machine-readable data model — in reference/02-schemas.md.

```
---
# === Identity (mandatory) ===
id: SPEC-<TYPE>-NN[.N]              # immutable; TYPE ∈
{ARCH,API,DATA,INT,PROC,UI,AI,SEC,OPS,TEST,DOC}
title: "<short, descriptive>"
type: SPEC-ARCH | SPEC-API | SPEC-DATA | SPEC-INT | SPEC-PROC | SPEC-UI | SPEC-AI
| SPEC-SEC | SPEC-OPS | SPEC-TEST | SPEC-DOC
slug: "<kebab-case>"                # auto-derived

# === Scope (mandatory) ===
level: system | subsystem | module
scope:
  system: "<system-id>"
  subsystem: "<subsystem-id>"       # null if level=system

# === Lifecycle (mandatory) ===
status: draft | review | approved | verified | obsolete
priority: must | should | could     # not all types use; mostly SPEC-SEC / SPEC-
OPS

# === Source: provenance (conditional, see chapter 7 §7.4.1) ===
# source.adapt — conditional (present when an ADAPT was created; §7.4.1.1).
# source.tz-section — always mandatory.
# source.adversarial-review-ref — mandatory when source.adapt is omitted.
source:
  adapt: ADAPT-NNN                  # conditional
  adapt-section: "Forward §N"       # mandatory if adapt is present
  tz-section: "§N.N"                # always mandatory
  adversarial-review-ref: AR-NNN   # mandatory if adapt is omitted (§7.4.6)

# === Link graph (auto-managed except mandatory ones) ===
referenced-by: []                   # auto-derived; SR/TR/SPEC referencing here
depends-on: []                      # mandatory if present; SPEC-* this SPEC
rests on
verified-by: []                     # auto-derived; list of verifying TC IDs

# === AI provenance (mandatory at RENAR-4+; canonical schema — §4.10.1) ===
ai-provenance:
```

```
  generated-by: "<vendor>-<model>@<date>"
  generated-at: "<ISO-8601>"
  prompt-template: "<template-path>@<version>"
  context-tokens: integer
  output-tokens: integer
  human-edits: boolean
  generation-time-ms: integer        # optional; see §4.10.1
  # optional at RENAR-4, mandatory at RENAR-5:
  # cost-budget, cost-actual

# === Replacement (mandatory if applicable) ===
replaces: "<old-id>"
replaced-by: "<new-id>"
deprecated-date: "<ISO date>"

# === Compliance (optional) ===
compliance-refs: []                # references to ISO/GDPR/AI Act/NIST AI RMF
---
```

8.4.1 Mandatory body sections

The body of any SPEC MUST contain:

1. **Purpose** — 1–3 paragraphs.
2. **Scope** — what is in, what is out.
3. **Type-specific sections** — see §8.5.
4. **Link to requirements** — which SR/BR reference it.
5. **Link to other SPEC** — `depends-on[]`.
6. **Verification** — which TC verify this SPEC.

8.5 Schema extensions by SPEC type

A brief description of type-specific fields and mandatory body sections. The full machine-readable extension schema — in reference/02-schemas.md. Industry references in detail — in the listed standards.

8.5.1 SPEC-ARCH

Type-specific frontmatter: `arch-style`, `deployment-model`, `tech-stack`, `quality-attributes`.

Mandatory body: system context (C4 L1), containers (C4 L2), components (C4 L3) for critical containers, quality attributes (latency / throughput / availability), ADR log.

Spec-specific TC (chapter 9): architecture conformance tests (zoning), reference tests of quality attributes.

8.5.2 SPEC-API

Type-specific frontmatter: `api-style` (rest / graphql / grpc / async-events), `api-version`, `versioning-strategy`, `authentication`, `rate-limits`, `contract-file` (location of machine-readable contract).

Mandatory body: endpoints / operations with payload / response / errors, versioning rules (breaking vs non-breaking), error model, authn/authz reference to SPEC-SEC, rate limits, 2–3 example requests per endpoint.

Spec-specific TC: contract tests, authentication negative, rate limit tests.

8.5.3 SPEC-DATA

Type-specific frontmatter: `data-style` (relational / document / graph / columnar), `storage-engine`, `schema-version`, `pii-classification[]`, `retention-policies[]`, `migration-strategy`.

Mandatory body: domain entities, ERD (text / Mermaid / link), entity fields (type / constraints / indices / defaults), relationships (FK / cardinality / cascade), PII / sensitive data classification + encryption at-rest + retention, migration approach, index strategy.

Spec-specific TC: migration tests, constraint tests (FK / NOT NULL / unique), PII handling tests, data retention tests.

8.5.4 SPEC-INT

Type-specific frontmatter: `integration-pattern` (request-response / event-driven / message-queue / webhook / file-transfer), `direction`, `counterparty`, `sla`, `idempotency`.

Mandatory body: integrated systems, exchange contract, failure modes + retry strategy, idempotency + dedup, security between systems, observability (correlation IDs).

Spec-specific TC: contract tests with a counterparty mock, failure injection, idempotency, end-to-end TC `tc-type: contract`.

Note: SPEC-INT replaces the existing `INT-SR` (§8.7 migration).

8.5.5 SPEC-PROC

Type-specific frontmatter: `process-style` (bpmn / state-machine / saga / choreography / orchestration), `state-count`, `participants[]`, `sla` end-to-end and per step, `compensation` (defined / not-applicable / manual).

Mandatory body: process diagram (BPMN-flavor / Mermaid / link), states and transitions (for a state machine), participants and their roles, happy path, alternative scenarios and exceptions, timeouts and compensation (for saga), SLA.

Spec-specific TC: happy path E2E, alternative paths, compensation tests (for saga), SLA tests.

8.5.6 SPEC-UI

Type-specific frontmatter: `ui-platform`, `target-users[]` (with references to ADAPT persona sections), `design-system`, `accessibility-level` (WCAG-A / AA / AAA), `i18n`, `mockup-links[]`, `baseline-images[]` for VLM-judge tests.

Mandatory body: overall interface structure, key screens, user journeys without technical details, cross-cutting elements (access rights / notifications / error / empty states), tone and style, accessibility, i18n.

Spec-specific TC: VLM-judge against a baseline (judge ≠ production isolation), accessibility (axe-core / Pa11y), i18n (string overflow / RTL), user journey E2E.

Note: SPEC-UI replaces the existing `UIC` (§8.7 migration).

8.5.7 SPEC-AI

Type-specific frontmatter: `ai-pattern` (rag / fine-tuning / prompt-engineering / tool-use / multi-agent), `production-model` (vendor / model / version), `judge-model` (MUST differ from production), `context-strategy`, `eval-strategy` (metric / threshold / baseline-dataset), `cost-budget`.

Mandatory body: AI component architecture (pipeline / orchestration / fallback), model card (capabilities / limits / known failure modes), context strategy, eval strategy with judge ≠ production isolation, cost management, hallucination mitigation, adversarial aspects.

Spec-specific TC: eval against a baseline (judge isolated), adversarial (prompt injection as a negative TC), cost regression, hallucination tests.

Note: SPEC-AI replaces the existing `AIC`. Isolation judge ≠ production model is a mandatory requirement of the standard for all eval-TC.

8.5.8 SPEC-SEC

Type-specific frontmatter: `security-domains[]`, `auth-model` (authn / authz strategies), `data-classification[]` (PII-high / PCI / internal), `threat-model-method` (STRIDE / PASTA / OCTAVE), `compliance[]`, `incident-response` reference in SPEC-OPS.

Mandatory body: auth model (authn flow / authz rules), data classification with protection, threat model (STRIDE table with a mitigation for each threat), secrets management, audit (what is logged / retention / access), encryption (at-rest / in-transit / key management), compliance mapping (references to specific clauses).

Spec-specific TC: authn (pos + neg), authz (RBAC matrix), threat-test (each STRIDE threat → at least 1 negative TC), audit log, secrets leakage.

8.5.9 SPEC-OPS

Type-specific frontmatter: `deployment-style`, `environments[]` (dev / staging / prod with purpose and scale), `slo`, `observability` (logs / metrics / traces / alerting), `runbook-link`, `disaster-recovery` (rto / rpo / backup-strategy).

Mandatory body: environments, deployment process (CI/CD pipeline / gating / rollout strategy), SLO (availability / latency / error budget), observability, alerting (critical alerts / escalation), runbook, capacity planning, disaster recovery.

Spec-specific TC: deployment tests (smoke), SLO regression (load testing), failover (DR drills), observability (alerts fire when expected).

8.5.10 SPEC-TEST

Type-specific frontmatter: `benches[]` (bench topology: nodes, versions, what is live / what is emulated), `counterparties[]` (emulators and sandbox instances of external systems — one entry

per integration), `datasets[]` (a dataset on both sides of the integration: volume, provenance, `anonymized: boolean`, `contains-real-client-data: boolean`), `reconciliation-rules[]` (rules for reconciling results against the data of foreign systems), `run-config` (what is mocked, what is live, the run order), `client-signature` (**mandatory** if at least one dataset carries the client's real or personal data).

Mandatory body: bench topology; the list of external systems and the way each is represented (real / sandbox / emulator — with rationale); datasets on both sides of every integration; reconciliation rules (**where the expected result lives**, if it is not in our system); run configuration; the data policy (volume, anonymization, retention period).

Spec-specific TC: reproducibility (a repeated run on the same bench yields the same result); counterparty validity (the sandbox answers like the real system — otherwise the proof is fictitious); dataset integrity (the data conforms to the declared schema and volume).

Link to TC. Every TC with `automation.kind: dynamic` MUST carry an `environment-ref` to a SPEC-TEST (§9.3): "the test passed" is meaningful only against a concrete bench and concrete data. Static checks (the doc-lint of §9.8, the structural TC of §9.8.1) require no bench — the analyzer works over artifacts — and the field does not apply to them. A change of bench or dataset increments the SPEC-TEST version and, per §10.5.4, invalidates `verified` on the dependent TC — **precisely**, without touching anything else.

The client signature. Volume, anonymization, and the admissibility of using the client's real data are a matter for **the client's** decision, not the contractor's. When `contains-real-client-data: true` or anonymization is incomplete, `client-signature` is mandatory.

8.5.11 SPEC-DOC

Type-specific frontmatter: `deliverables[]` (one entry per document: `kind` — user manual / administrator manual / training materials / other; `audience`; `format`; `language`), `required-sections[]` (the mandatory sections of each document — **set as a requirement**, not left to the contractor's discretion), `traces-to[]` (the BR / SR / SPEC whose behavior the document describes), `version-binding` (the rule matching the document version to the system version), `acceptance-criteria[]` (against what the document is accepted).

Mandatory body: the composition of the delivery (which documents and for which audience); the structure of each document; the requirements on it (completeness across the roles and scenarios declared in the requirements; version currency; language and format); the readiness criteria.

Spec-specific TC: a **doc lint** (§9.8) — mandatory.

Why the doc lint is mandatory. A non-verifiable SPEC does not pass QG-2 (§9.7, MVR-5 requires coverage of every normative statement). A SPEC-DOC without a machine check would either block the delivery or force a hole in MVR-5 for the sake of a single type. Its statements are therefore normed checkably: the presence of the mandatory sections, coverage of all roles and scenarios from the linked BR / SR, the match of the document version to the system version, format and language — all of this is checked mechanically (`automation.kind: static`). Substantive conformance to the implemented behavior ("the text does not lie about the system") is checked optionally by a judge agent through the already existing P7 / `eval` mechanism — no new entities are required.

8.6 Link to requirements and tasks

8.6.1 SR.constrained-by[]

An SR receives a `constrained-by[]` field in its frontmatter — typed references to SPEC. This is a **graph**, not a tree of parents. The SR's parent in the requirements tree is single (BR).

```
# SR frontmatter (example)
id: SR-05
parent:
  id: BR-02
constrained-by:
  - SPEC-UI-01
  - SPEC-API-02
  - SPEC-DATA-03
  - SPEC-PROC-01
  - SPEC-SEC-01
verified-by:
  - TC-12
  - TC-13
source:
  adapt: ADAPT-001
  adapt-section: "Forward §3"       # see canonical identifier §8.4
```

8.6.2 TR.implements-spec[]

A TR (task) references its SR (parent in the tree) + one or more SPEC through `implements-spec[]`:

```
id: TR-42
title: "Implement endpoint POST /orders"
parent:
  id: SR-05
implements-spec:
  - SPEC-API-02
  - SPEC-DATA-03
verified-by:
  - TC-14
```

8.6.3 SPEC.depends-on[]

A SPEC MAY rest on another SPEC:

```
id: SPEC-API-02
title: "Orders REST API"
type: SPEC-API
depends-on:
```

```
  - SPEC-DATA-03          # stable data schema
  - SPEC-SEC-01           # auth model for endpoints
```

When an upstream SPEC changes (for example SPEC-DATA-03), all downstream artifacts (SPEC-API-02 and the SR linked through it) MUST be reviewed: either `verified` is reconfirmed (the change is compatible), or the downstream artifact passes re-verification through its own state machine (§10.7) and, until that completes, is not considered `verified` with respect to the new upstream version.

8.6.4 Auto-derived inverse edges

`SPEC.referenced-by[]` is recomputed by a substrate hook after each change to SR / TR / SPEC. An orphan SPEC (without `referenced-by[]` and without an active status) is a warning in the quality report.

8.7 Migration UIC / AIC / INT-SR / TS → SPEC-*

8.7.1 Mapping table

Old type	New type	Migration kind
`UIC-NN`	`SPEC-UI-NN`	Rename ID + move to `specs/ui/`
`AIC-NN`	`SPEC-AI-NN`	Rename ID + move to `specs/ai/`
`INT-SR-NN`	`SPEC-INT-NN`	Rename ID + move to `specs/int/`
`TS-NN`	`SPEC-<TYPE>-NN` (distribution)	Manual review of each TS; an AI agent classifies the content, the architect approves in one click

8.7.2 Atomic migration

Migration is one atomic change unit (V2) at the project level. Parallel existence of the old types (UIC / AIC / INT-SR / TS) and SPEC-* as the source of truth is prohibited.

Procedure (substrate-independent):

1. Preparation: an AI agent classifies each existing TS-NN into one of the 11 SPEC types.
2. The architect approves the classification.
3. Atomic change: rename IDs (UIC→SPEC-UI; AIC→SPEC-AI; INT-SR→SPEC-INT; TS→SPEC-*), move files to `specs/<type>/`, update all references in BR / SR / TR / TC frontmatter (`parent: UIC-NN` → `constrained-by: [SPEC-UI-NN]`).
4. Regeneration of auto-derived files (REQUIREMENTS.md, SPECS.md, inverse edges).
5. CI check: absence of orphan references and old IDs.

8.7.3 ID immutability

After migration, SPEC IDs are **immutable** (see V1, §3.3.1). Renaming `SPEC-API-02` → `SPEC-API-08` is prohibited. Replacement is through `deprecated` + a new ID with `replaces[]`.

8.8 Quality gates for SPEC

A SPEC has a dedicated state machine (chapter 10 §10.7):

State	Transition condition
`draft`	Created; mandatory frontmatter fields are being filled
`review`	Ready for review; mandatory body sections (§8.4.1) and type-specific ones (§8.5) are present
`approved`	The architect confirmed; `depends-on[]` consistency checked
`verified`	All mandatory spec-specific TC (chapter 9 §9.7) are green
`obsolete`	Replaced or no longer relevant; `replaced-by` is mandatory

Link to QG-0 / QG-2 of SENAR:

- QG-0 (the task has a goal/AC) is extended: for tasks implementing a SPEC, `implements-spec[]` in the TR frontmatter is mandatory.
- QG-2 (a `done` task has evidence) is extended: for tasks implementing a SPEC, a TC of the corresponding spec-specific kind (chapter 9 §9.7) is mandatory.

8.9 Storage layout

8.9.1 At the system level

```
[requirements-substrate]/      # root of the requirements substrate (layout —
guide/03 or guide/04)
  br/
  sr/
  specs/
    arch/   SPEC-ARCH-NN-*.md
    api/    SPEC-API-NN-*.md
    data/   SPEC-DATA-NN-*.md
    ui/     SPEC-UI-NN-*.md
    ai/     SPEC-AI-NN-*.md
    int/    SPEC-INT-NN-*.md
    proc/   SPEC-PROC-NN-*.md
    sec/    SPEC-SEC-NN-*.md
    ops/    SPEC-OPS-NN-*.md
    test/   SPEC-TEST-NN-*.md
    doc/    SPEC-DOC-NN-*.md
  adapt/
```

```
  tz/
  SPECS.md                # auto-generated index
```

> *All 11 SPEC types (§8.3) are allowed at any `level`; the `specs/<type>/` subfolders are created as needed — not all are mandatory at the system level.*

8.9.2 At the subsystem level

```
[subsystem-substrate]/         # subsystem scope
  br/                     # if it has its own business side
  sr/
  specs/
    arch/   SPEC-ARCH-NN-*.md      # subsystem architecture
    api/    SPEC-API-NN-*.md
    data/   SPEC-DATA-NN-*.md
    ui/     SPEC-UI-NN-*.md
    ai/     SPEC-AI-NN-*.md
    int/    SPEC-INT-NN-*.md
    proc/   SPEC-PROC-NN-*.md
    sec/    SPEC-SEC-NN-*.md
    ops/    SPEC-OPS-NN-*.md
    test/   SPEC-TEST-NN-*.md
    doc/    SPEC-DOC-NN-*.md
  adapt/
  SPECS.md
```

The substrate-native storage implementation is substrate-specific (see guide/03, guide/04).

8.9.3 SPECS.md — auto-generated index

`SPECS.md` is an auto-generated registry of all SPEC: ID, type, title, status, link to the verifiable requirement, link to the file. Marked `linguist-generated=true`. Regeneration triggers — every change to SPEC frontmatter or every approve / verify gate.

8.10 Links to other chapters

Chapter	Link
02 Methodology positioning	SPEC as a parallel axis — a consequence of the Source-of-Truth inversion
06 Requirements hierarchy	`SR.constrained-by[]`, `TR.implements-spec[]`
07 ADAPT	SPEC references ADAPT through `source.adapt`
09 Test cases	Spec-specific TC types (the table of mandatory TC kinds for each SPEC type)

10 Lifecycle and QG	SPEC state machine + QG extensions for SPEC
03 Substrate versioning	SPEC IDs are immutable (V1); migration is atomic (V2)
11 Maturity model	RENAR-3+: all 11 SPEC types where applicable
reference/02 — schemas	Full machine-readable schema for each type-specific extension
reference/05 — knowledge graph schema	`constrained-by[]`, `implements-spec[]`, `depends-on[]` as edge types in the graph

09. Test Cases

> ***Part of the RENAR Standard v1.0*** · *← Table of contents*

9.1 A test case is a first-class artifact

In RENAR a test is not a postscript at the end of the code, but a full-fledged document: it has its own version, status, and place in the trace chain, just like the requirement it verifies. The reason is simple: tests are written by an AI agent, and an AI happily covers the "happy path" ("entered the right password — let in") and quietly skips the unpleasant parts ("entered someone else's — MUST NOT let in, MUST NOT hint which field is wrong, MUST NOT write the password to the log"). It is precisely in the unhandled negative cases that defects live.

For this reason RENAR makes two requirements normative. **Pos/neg pairing**: for every verifiable statement — at least one positive test case and one negative one (what MUST happen and what MUST NOT happen). **Judge isolation**: if the result is assessed by another AI model (for the `ux`, `eval` types), it MUST differ from the one that produced the artifact under assessment — a model does not check itself. The TC (Test Case) closes the trace chain TZ → ADAPT → BR / SR / SPEC → TR → TC (see §2.3): from a test failure one can trace back to the TZ section it ultimately verifies.

The chapter builds on ISO/IEC/IEEE 29119 "Software testing" for the concepts of test design, test execution, test result reporting, and pos/neg coverage, but fixes a closed list of TC types, mandatory pos/neg pairing, and judge ≠ production isolation as normative requirements of v1.0 that are not present in formalized form in ISO 29119.

From the practices of Specification by Example (Adzic) and BDD / Gherkin — where executable examples also serve as a specification — RENAR differs in that it moves pos/neg pairing (§9.7), judge ≠ production isolation (§9.13), and version-pinning a TC to the requirement version (V5, §3.3.5) from a recommendation to blocking normative clauses (§14.5.2).

The clauses of this chapter are normative. The closed lists (principles, TC types, mandatory TC kinds per SPEC type) are RENAR mandatory clauses (chapter 13); extension is only through the formal change procedure of the standard.

> ***Dense chapter:*** *reference/09 · the decision tree below is informative routing.*

9.1.1 Decision tree: choosing `tc-type` (informative)

```
flowchart TD
  A[New TC for SR/SPEC] --> B{SPEC type?}
  B -->|SPEC-UI| C[tc-type: ux]
  B -->|SPEC-AI| D[tc-type: eval + adversarial negative]
  B -->|SPEC-API / INT / DATA| E[tc-type: contract]
  B -->|SPEC-SEC| F[tc-type: security — negative only]
  B -->|SPEC-ARCH / PROC / OPS| G[tc-type: system]
  B -->|BR business goal| H[tc-type: business]
  C --> I{pos/neg pair?}
  D --> I
```

```
E --> I
F --> J[security: negative mandatory; positive via system TC]
G --> I
H --> I
I -->|SR normative claim| K[§9.7: pos + neg required]
I -->|negative invariant only| L[§9.7 exception: single TC OK]
```

The adversarial-review procedure for TC — guide/07 §3.5; the agent panel (informative) — same place, §4.5.

9.2 Closed list of normative TC principles

#	Principle	Normative formulation
P1	First-class artifact (TC)	A TC is a standalone artifact of the standard, equal to a requirement in lifecycle and versioning. It is stored as a separate file in the `tests/` subfolder of the requirements substrate (§9.17).
P2	Document ≠ implementation	A TC describes **what is verified and how** (decoupled from the implementation). The implementation (code) is addressed by the `automation.location` field and stored in the code substrate. One TC — one implementation.
P3	AI-generated	TC are created and edited by an AI agent on the engineer's assignment; the engineer does not write TC by hand (see chapter 11 §11.N).
P4	AI-executed	All TC in status `ready` and above are automated. The run is performed by an automated runner (CI / AI-runner / specialized executor). Results in `last-run` are recorded only by the runner (a bot-managed actor) upon the run.
P5	Pos/neg pairing	For every statement of a requirement (BR / SR / SPEC), at least one positive + negative TC pair is created (§9.7).
P6	`last-run` — bot-managed	The `last-run` field (date / result / runner-id / requirement-version / judge-report) is filled in only by the automated runner. Manual editing of `last-run` by any actor is prohibited by the standard (§9.12).
P7	Judge ≠ production isolation	For TC types that use LLM-as-judge (ux, eval), the judge model **MUST NOT coincide** with the production model that generates the artifact under assessment. A coincidence is blocked by a substrate hook (§9.6.2).
P8	Test authorship isolation	A test is separated from the implementation along three axes: **time** — the test is frozen (`ready`) before the implementation it verifies is started; **author** — the agent writing the test is not the agent writing the implementation (§9.18); **change** — an edit to the criteria of a frozen test is made only through `[test-spec-change]` with an engineer's approval (§9.13).
P9	Red history	A test that has never been observed red **is not evidence**. The fixing run is performed before the implementation: the test MUST be red; the "red → green" transition is recorded (§9.18).

The list is closed. New principles are added only through the formal change procedure of the standard (chapter 13).

Why P9 is needed when P8 exists. Authorship isolation guarantees that the test was written **before** the code and **not by** whoever writes the code. It does not guarantee that the test **verifies anything**: an empty test, honestly written by an isolated agent by mistake, is formally impeccable and green from birth — it passes all three axes of P8. Red history closes exactly this residue, and nothing else closes it.

9.3 General TC schema (frontmatter)

All TC types share a common set of frontmatter fields. Type-specific fields are added as extensions on top (§9.6). The full machine-readable schema — in reference/02-schemas.md.

```
---
# === Identity (mandatory) ===
id: TC-NN                              # immutable; NN sequential within scope
title: "<short, descriptive>"
type: TC
slug: "<kebab-case>"                   # auto-derived

# === Classification (mandatory) ===
tc-type: business | ux | system | contract | eval | security
negative: boolean                      # true for the paired negative TC

# === Scope (mandatory) ===
level: system | subsystem | module
scope:
  system: "<system-id>"
  subsystem: "<subsystem-id>"          # null if level=system
  module: "<module-id>"                # null if level ≠ module

# === Lifecycle (mandatory) ===
status: draft | ready | passing | failing | obsolete

# === Verification target (mandatory; at least one) ===
verifies:
  - id: SR-NN | BR-NN | SPEC-<TYPE>-NN
    requirement-version: "<substrate-native version-ref>"   # V5 pinning (see
chapter 3)
  - id: ...

# === Pair link (mandatory if negative=false and a pair exists) ===
paired-with:                           # ID of the paired TC (positive ↔ negative)
  - TC-NN

# === Task binding (optional; §9.19.7) ===
verifies-tr: TR-NN                     # the task to whose scope the test is
narrowed
verifies-claims: []                    # subset of the parent SR's statements
covered within the TR scope

# === Red history (mandatory for a TC to count as evidence; §9.18.2) ===
red-history:
  fixing-run: { date: "<ISO-8601>", result: fail }     # the fixing run BEFORE
```

```
implementation MUST be red
  green-transition: "<ISO-8601>"                          # the red → green
transition (runner-managed)
  inherited-from: TC-NN                                   # conditional: task with
no behavior change (refactoring)
  not-applicable-reason: implementation-originated        # conditional: class
§6.13; REQUIRES mutation-check below
mutation-check:                                           # mandatory if not-
applicable-reason is set (§6.13.3)
  mutants-killed: integer                                 # MUST be ≥ 1

# === Bench and data (conditional; §8.5.10) ===
environment-ref: SPEC-TEST-NN        # mandatory if automation.kind: dynamic —
the bench and dataset
                                     # the TC is valid against. Does not apply to
static (the doc lint
                                     # §9.8, a structural TC §9.8.1): the
analyzer works over the
                                     # artifacts, not against a bench

# === Automation (mandatory) ===
automation:
  status: automated | manual-pending
  kind: dynamic | static             # mandatory; static — the runner is a static
analyzer (§9.8.1)
  location: "<substrate-native pointer to implementation>"  # mandatory if
automated
  manual-pending-until: "<ISO date>"                         # mandatory if
manual-pending
  manual-pending-reason: "<text>"                             # mandatory if
manual-pending

# === Execution (mandatory if type=ux | eval) ===
judge:
  vendor: "<provider>"               # mandatory; see P7 isolation
  model: "<model-id>"
  prompt-template: "<template-path>@<version>"

baseline:                            # mandatory for ux | eval
  artifact: "<substrate-native pointer>"
  perceptual-diff-threshold: float   # for ux
  metric-thresholds: {}              # for eval

# === Last run (auto-managed; bot-only) ===
last-run:
  date: "<ISO-datetime>"
  result: pass | fail | skipped | n/a
  runner-id: "<runner-name@version>"
  run-ref: "<substrate-native reference>"
  requirement-version: "<version-ref of verified artifact>"
  judge-report: "<inline or pointer>"

# === AI provenance (mandatory at RENAR-4+; canonical schema — §4.10.1) ===
ai-provenance:
```

```
  generated-by: "<vendor>-<model>@<date>"
  generated-at: "<ISO-8601>"
  prompt-template: "<template-path>@<version>"
  context-tokens: integer
  output-tokens: integer
  human-edits: boolean
  # optional at RENAR-4, mandatory at RENAR-5 (see §4.10.1):
  # cost-budget, cost-actual, generation-time-ms

# === Replacement / obsolescence ===
obsolete-pending: boolean            # true on detected delta-TZ invalidation
replaces: "<old-id>"
replaced-by: "<new-id>"
obsoleted-date: "<ISO date>"
---
```

`verifies[]` is a closed list of references to verifiable artifacts (BR / SR / SPEC). TR is not stated directly in `verifies` — a TR is verified through its parent SR (see §6.7). `verifies[].requirement-version` is the substrate-native pinning of the artifact (V5 capability, see chapter 3 §3.3.5); QG-2 (§9.10) requires `verifies[].requirement-version` to match the current version of the artifact.

9.4 TC body sections

The body of any TC mandatorily contains the following sections (regardless of substrate):

Section	Obligation	Content
Context	mandatory	Which clause of the verifiable artifact the TC references; a quotation or paraphrase of the statement.
Preconditions	mandatory	The state of the system and data required for the run; provided by a seed mechanism.
Steps	mandatory	Runner actions; for `tc-type: ux` — intents, not selectors (see §9.6.1).
Pass criterion	mandatory	Binary, observable, reproducible (see §9.11).
Fail criterion	mandatory	A list of observable signs of a violation (not the negation of Pass); includes leaks, side-effects, race conditions.
Postconditions	mandatory	What state is expected after the run; cleanup mechanism.
Out of scope	mandatory	What is **intentionally** not verified, with a reference to the paired TC where it is covered.
Related TC	optional	References to semantically related TC.

The "Out of scope" section is normatively mandatory: it guards against a false sense of coverage. The absence of the section blocks the TC's transition into `ready`.

The body section names are machine-detectable `##`-level headings. The canonical section identifiers for the criteria sections are `## Pass criterion` and `## Fail criterion`; it is precisely these that the change-of-criteria control hook detects (§10.11.3), so the names of these sections are fixed and not subject to local substitution.

9.5 Closed list of TC types

```
tc-type ∈ { business, ux, system, contract, eval, security }
```

Type	What it verifies	Applied to	runner family
`business`	Is the business goal achieved?	BR	E2E + AI validator
`ux`	Does the UX match the stated experience?	SPEC-UI	AI-driver + VLM-judge
`system`	Does the system behave as described?	SR, SPEC-PROC, SPEC-ARCH	xUnit family
`contract`	Is the contract honored?	SPEC-API, SPEC-INT, SPEC-DATA	Contract-testing framework
`eval`	Is the AI component quality achieved?	SPEC-AI	Eval-runner with a reference dataset
`security`	Are the security invariants honored?	SPEC-SEC	Authz/threat-test framework

The list is closed. New types are added only through the formal change procedure of the standard (chapter 13). Specific runner technologies are substrate-specific and fixed in the implementation conformance manifest.

Why `business` and not `acceptance`. RENAR has two acceptances, on opposite sides of the contour decoupling: a `business` TC verifies that the business goal of a BR is achieved (the internal contour), while an **AT** (§9.19) verifies conformance to the contract, that is, to the effective TZ. One name for two different subjects is a direct source of confusion; the names are kept apart.

9.6 Type-specific extensions

9.6.1 `tc-type: ux` — UX tests based on SPEC-UI

A UX test is normatively built as a two-layer structure:

Layer	Content	Executor
Scenario (intent)	" wants after "	AI-driver: translates the intent into actions, finds elements by semantics (without hard selectors)

Perceptual check	"On the rendered state, is visible"	Perceptual judge (VLM): takes the render + criterion, returns pass/fail with a rationale

Mandatory frontmatter extension: `judge.vendor`, `judge.model`, `baseline.artifact`, `baseline.perceptual-diff-threshold`.

Mandatory body sections in addition to §9.4: Scenario (intent, not selectors); Perceptual criterion (what the judge MUST see); Paired negative (empty state / error / lack of permissions).

Visual regression. In addition to the VLM-judge — a perceptual diff against `baseline.artifact` with the threshold `perceptual-diff-threshold`. Exceeding the threshold blocks the TC's transition into `passing`.

Baseline update. Changing `baseline.artifact` REQUIRES a substrate-native approval mechanism with the tag `[baseline-update]` (see §9.13); automatic update is prohibited.

9.6.2 `tc-type: eval` — Eval tests based on SPEC-AI

An eval test normatively verifies the quality of an AI component through a dataset with metrics and thresholds.

Mandatory frontmatter extension: `judge.vendor`, `judge.model`, `baseline.artifact` (versionable dataset), `baseline.metric-thresholds`.

Mandatory body sections: dataset provenance (how it was assembled, what labeling was applied); metric cluster (one eval-TC = one semantically coherent group of metrics; different families — different TC); regression rule (what counts as a failure — going outside the threshold or a regression of ≥ N% against the baseline).

Judge ≠ production isolation (normative). The `judge.vendor` + `judge.model` field MUST differ from the `production-model` of the SPEC-AI specification that normalizes the behavior under assessment. A substrate-native hook (chapter 3 §3.3.3) MUST block the merge of a change unit on a coincidence.

Dataset versioning. The eval dataset is a substrate-managed versionable artifact: every change is recorded as an atomic change unit with a description (what was added / removed / re-labeled) and authorship (generator agent / critic agent / human spot-check).

Cost gating. Eval is not run on every implementation change (cost): the runner is triggered on a change to SPEC-AI, the production model, or the dataset, or on a schedule. The triggers are fixed in the substrate-native runner configuration.

Two-stage dataset labeling. The generator agent creates candidates; the critic agent checks them against a checklist; the engineer performs a spot-check of ≥ 10% of random examples before merging the dataset.

9.6.3 `tc-type: contract` — Contract tests based on SPEC-API / SPEC-INT / SPEC-DATA

Mandatory body sections: machine-readable contract (a reference to OpenAPI / GraphQL SDL / Protobuf / JSON Schema from the SPEC); producer side / consumer side; mocked counterparty (for SPEC-INT — sandbox / real environment as a separate TC).

Mandatory extension for SPEC-INT. Contract TC MUST be combined with an integration TC (`tc-type: contract` , `level: subsystem | system`) against a real or sandbox counterparty — a mocked contract is not sufficient to verify SPEC-INT.

9.6.4 `tc-type: security` — Security tests based on SPEC-SEC

Mandatory body sections: threat-model attributes (STRIDE category or equivalent); subject under test (authn / authz / data classification / secrets / audit / encryption); negative scenarios (bypass attempt, unauthorized access, leakage); expected system behavior on a violation.

A security TC normatively contains **only negative scenarios** (an attempt to bypass protection). The positive "grant correct access to the correct actor" is covered by `tc-type: system` with coverage scope SPEC-SEC.

9.7 Pos/neg pairing — normative requirement

For every statement of a requirement (BR / SR / SPEC) that describes observable behavior, at least one positive + negative TC pair is created.

Positive TC	Paired negative TC
`negative: false`	`negative: true`
Describes the happy path / success behavior	Describes boundary conditions, violations, bypasses
`paired-with: [TC-<neg-id>]`	`paired-with: [TC-<pos-id>]`

A negative TC normatively describes the observable signs of a violation (what MUST NOT happen), not the negation of the positive TC's Pass criterion. Examples:

Statement	Pos TC	Neg TC
"Authentication by email + password"	Correct credentials → 200 + JWT	Wrong password → 401 without disclosing which field is wrong; no record in the session-store; rate-limit after N attempts
"Order creation"	Valid payload → 201 + order-id	Invalid price < 0 → 422 with an explicit error; no record in the DB; no side-effect (notification, accrual)

QG-2 (§9.10) MUST block the promotion of an artifact to `verified` if at least one normative statement is covered only by a positive TC.

Single-TC coverage is permitted **only** in one case: the artifact describes a prohibition / negative invariant on its own (for example, a security TC by STRIDE category — it is negative by nature).

9.8 Spec-specific TC — mandatory kinds per SPEC type

A closed normative table: each SPEC type MUST have at least one TC of each "mandatory kind" before transitioning into `verified` (chapter 8 §8.8).

SPEC type	Mandatory TC kinds	Additional TC kinds
SPEC-ARCH	Conformance (zoning / dependency rules)	Reference quality-attribute values (latency / throughput / availability)
SPEC-API	Contract (against the contract from `contract-file`)	Auth negative; rate-limit; versioning compatibility
SPEC-DATA	Constraint (FK / NOT NULL / unique); Migration (forward pass + rollback)	PII handling; retention; index regression
SPEC-INT	Contract (mocked counterparty); contract TC `tc-type: contract` (real / sandbox counterparty)	Failure injection; idempotency; observability (correlation IDs)
SPEC-PROC	Happy path E2E; Alternative paths E2E	Compensation (for saga); SLA end-to-end
SPEC-UI	VLM-judge against a baseline (judge ≠ production); Accessibility (WCAG-AA minimum)	i18n (string overflow / RTL); Journey E2E
SPEC-AI	Eval against a reference dataset (judge isolated)	Adversarial (prompt injection as a negative TC); Cost regression; Hallucination tests
SPEC-SEC	Authz / RBAC matrix; Threat-test per STRIDE category	Audit log; Secrets leakage; Encryption invariants
SPEC-OPS	Smoke after deploy; SLO regression (load test)	Failover / DR drill; Observability (alert firing correctness)
SPEC-TEST	Reproducibility (a repeat run on the same bench yields the same result); Counterparty validity (the sandbox answers as the real system does); Dataset integrity (the data conform to the declared schema and volume)	Bench drift (the configuration has not diverged from the declared one); Completeness of anonymization
SPEC-DOC	**Doc lint** (`automation.kind: static`): the presence of the mandatory sections; coverage of all roles and scenarios from the linked BR / SR; the match of the document version to the system version; the language and format	Substantive conformance to the implemented behavior — by a judge agent through P7 / `eval`

The substrate-native hook `promote SPEC → verified` (chapter 3 §3.3.3) MUST check for the presence of at least one TC of each mandatory kind and block the transition in their absence.

Self-reference of SPEC-TEST. The TC that verify the `SPEC-TEST` itself (reproducibility, counterparty validity, dataset integrity) are run **on the very bench** they describe, and therefore carry an `environment-ref` to that same bench. The circle is admissible and is the norm: the bench proves its own fitness in the only way available — by working. Invalidation remains correct even so: a change of bench increments its version and devalues its own TC among the rest — which is exactly what is required.

The table is closed at v1.0. Extension is only through the formal change procedure of the standard (chapter 13).

SPEC-DOC — the doc lint (mandatory). The checkable statements of the delivered documentation (§8.5.11): the presence of the mandatory sections; coverage of all roles and scenarios declared in the linked

BR / SR; the match of the document version to the system version; the language and format of the delivery. The runner is a static analyzer (`automation.kind: static`). Substantive conformance to the implemented behavior is checked optionally by a judge agent through P7 / `eval` — no separate mechanism is introduced.

Without the doc lint a SPEC-DOC is non-verifiable: QG-2 requires coverage of every normative statement (§9.7), and such a SPEC would either block the delivery or force a weakening of MVR-5 for the sake of a single type.

SPEC-TEST — reproducibility, counterparty validity, dataset integrity (§8.5.10). Counterparty validity is no formality: a sandbox that answers unlike the real system turns the proof into a fiction.

9.8.1 A structural TC — the degenerate normative part

For SPEC-ARCH and SPEC-DATA the mandatory TC kind is structural conformance (zoning and dependency rules, schema conformance). Its runner is a **static analyzer**, not a run of the system; this is recorded by the field `automation.kind: static` (§9.3). No separate TC type for structural checks **is introduced**: the check is already mandatory — all that was missing was the acknowledgement of the nature of its runner.

> *In a structural TC the normative part is **degenerate**: the statement is already written in the SPEC-ARCH / SPEC-DATA; the normative part of the test merely points at which clause of the specification is subject to automated checking.*

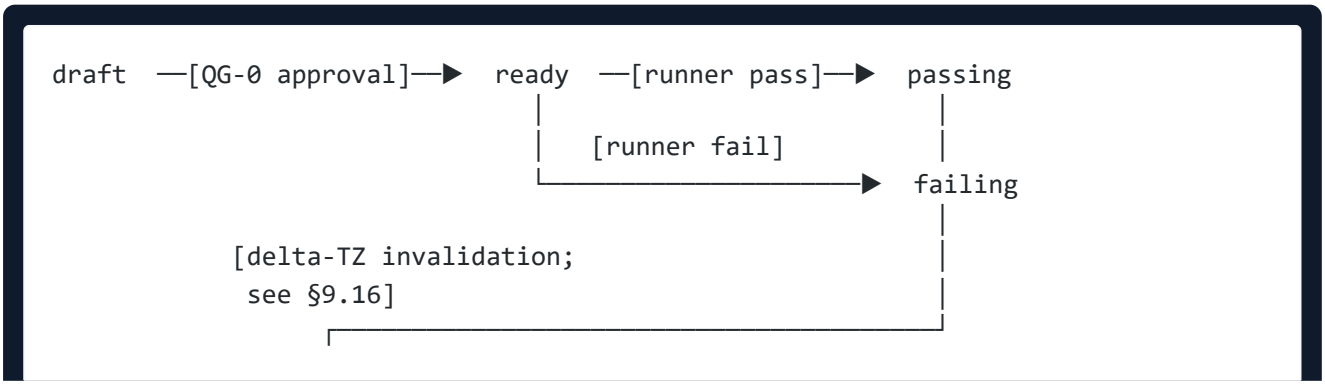
Degeneracy dictates the **failure-triage route** — otherwise the argument "what do we fix: the code, the test, or the specification?" is unavoidable:

Cause of failure	What changes
The code violates a rule written in the SPEC	The **code**
The rule is judged wrong or in need of tightening	The **SPEC** — a new version (not an edit to the test!)
The analyzer yields a false result	The **tool**; the baseline is untouched

Editing a structural TC in order to turn a failure green is test-gaming in its purest form: the baseline lives in the SPEC, and it is the SPEC that must change.

9.9 TC lifecycle

9.9.1 State machine



```
                    ▼
                obsolete
```

Status	Meaning	Transition trigger
`draft`	Created, implementation in progress	Creation by the AI agent
`ready`	Implementation valid; dry-run runner passed; pos/neg pairing confirmed	QG-0 (§9.10): one-click approval
`passing`	The latest run had `last-run.result = pass` on the current `requirement-version`	Bot-managed upon the run
`failing`	The latest run had `last-run.result = fail`	Bot-managed upon the run
`obsolete`	Terminal; the covered behavior no longer exists	Delta-TZ invalidation (§9.16) or deprecation of the parent artifact

`obsolete` is a terminal status. A TC in `obsolete` is substrate-natively preserved as a historical trace: the substrate implementation **MUST** ensure the immutability of the TC identifier and **MUST NOT** allow its reuse for a new TC (V1 capability; see chapter 3 §3.3.1).

9.9.2 Relation to the status of the verifiable artifact

Moving a BR / SR / SPEC into `verified` normatively requires: all TC from the verifiable artifact's `verified-by` have `last-run.result = pass` and `last-run.requirement-version` matches the current version of the artifact (see §9.10 QG-2).

9.10 Quality Gates for TC

The canonical definitions of the Quality Gates — in chapter 10 §10.3. This section is a TC-local summary of the gates applicable to TC directly (QG-0, QG-1) or using TC as evidence (QG-2). The numbering and semantics MUST match the canonical §10.3; a project-level local override is prohibited (§10.10.2).

Gate (canonical)	Role of TC	Precondition (brief; full formulation — ch. 10)	Postcondition
QG-0 — approval (§10.3.1)	TC (`draft → ready`) — the "approval" part	The `verifies[]` reference — the artifact exists in the substrate in a state no lower than `approved` ; the general preconditions of §10.3.1; the approver's decision is recorded substrate-natively (V3 + V6)	The TC is admitted to the implementation-gate checks (QG-1)
QG-1 — verification implementation (§10.3.2)	TC (`draft → ready`) — the "implementation" part	`automation.status = automated` (or `manual-pending` with a deadline and reason); pos/neg pairing (§9.7); dry-run runner passed; the mandatory TC body sections (§9.4) are filled in	The TC moves into `ready` ; the production runner run is admitted

QG-2 — verification (§10.3.3)	Artifact (BR / SR / SPEC / TR) `→ verified` / `→ done`; TC — evidence	All TC from the verifiable artifact's `verified-by` are in status `passing`; pos/neg pairing for each normative statement; the mandatory spec-specific TC kinds (§9.8) are present; `last-run.requirement-version` matches the current `version` of the artifact	The verifiable artifact moves into `verified` (a TR — into `done`); the TC stays `passing`

The substrate-native one-click promote `draft → ready` atomically checks the preconditions of both QG-0 and QG-1 as a single bundle (see §10.3.2 "Trigger") — the diagram in §9.9.1 shows the aggregate gate passage as "QG-0 approval".

Checking that `last-run.requirement-version` matches the current `version` of the verifiable artifact on every subsequent TC run is a runner-managed consistency check (§10.9.3), **not** a separate Quality Gate in the sense of §10.2.1. On a mismatch, the substrate automatically moves the TC into `failing` until a re-run on the current artifact version; the audit-trail record (§10.13) is recorded with the type `runner-fail`, not `gate-passage`.

The substrate-native hooks (chapter 3 §3.3) MUST block gate transitions that violate a precondition.

9.11 Pass / Fail / Out of scope — normative criteria

9.11.1 Pass criterion

The Pass criterion MUST be:

- **Binary** — yes or no, without interpretation.
- **Observable** — recorded without access to the system's internal structures.
- **Reproducible** — a repeated run under the same conditions yields the same result.

Bad	Good
"Login works correctly"	"`POST /auth/login` with valid credentials returns 200 and a JWT with `exp = now + 24h ± 1m`"
"Performance is acceptable"	"p95 latency < 200 ms at 100 RPS on `/search` over 5 minutes"
"The error is handled"	"On an invalid email, 422 is returned with body `{"field":"email","code":"invalid_format"}`"

9.11.2 Fail criterion

The Fail criterion is **not the negation** of Pass. It enumerates observable signs of a violation, including those the Pass criterion does not explicitly cover:

- Which response / state / event is recognized as a failure.
- Information leaks (for example, a 401 MUST NOT indicate exactly which credentials field is wrong).
- Side-effects that MUST NOT occur — a log record, an email being sent, mutation of other records.
- Race conditions: concurrent requests do not lead to a violation of invariants.

9.11.3 Out of scope

Every TC mandatorily contains an "Out of scope" section with an explicit enumeration of:

- What is intentionally not verified by this TC;
- Where it is covered (a reference to the paired TC or another TC).

The section guards against a false sense of coverage (see also chapter 11 — coverage matrix). The absence of an explicit "Out of scope" normatively blocks QG-0 (§9.10).

9.12 `last-run` — bot-managed only

The `last-run` field (date / result / runner-id / run-ref / requirement-version / judge-report) is filled in **only** by the automated runner (CI-bot / AI-runner / specialized executor). Manual editing of `last-run` by any human is a violation of the standard; the substrate hook (chapter 3 §3.3.6) MUST block a change unit that alters `last-run` from an author who is not a bot.

Composition of `last-run`:

Field	Mandatory	Content
`date`	yes	ISO-datetime of the run
`result`	yes	`pass | fail | skipped | n/a`
`runner-id`	yes	Runner identifier + version
`run-ref`	yes	substrate-native pointer to the full run log
`requirement-version`	yes	Version of the verifiable artifact at the time of the run (V5 pinning)
`judge-report`	yes for `ux | eval`	Inline or a pointer to the VLM/eval-judge report

---

9.13 Protection against test gaming

9.13.1 Normative rule

An AI agent MUST NOT simultaneously change the implementation code and the Pass / Fail criteria of an existing TC in a single change unit such that a `failing` TC becomes `passing`, without an explicit approval by the engineer of the TC change.

Without this rule, the AI agent has a trivial path to a green run — to weaken the criterion instead of fixing the code.

9.13.2 The `[test-spec-change]` mechanism

Change class	Tag	Approval

Change to the Pass / Fail criteria of an existing TC	`[test-spec-change]`	Mandatory: an explicit engineer approval of the change unit, separate from any implementation-code change
Change to `automation.location` (relocating the implementation without changing the verified behavior)	—	Without a separate approval
Change to the implementation code without TC edits	—	Standard workflow
Update of `baseline.artifact` / dataset / `mockup-baseline`	`[baseline-update]`	Mandatory: an explicit engineer approval
Creation of a new TC	—	QG-0 (§9.10)

The substrate-native implementation of the tags is substrate-specific (see guide/03, guide/04); the normative requirement is the atomicity of the change unit, an explicit designation of the change class, and the optional (but recommended) isolation of such changes into a separate change unit from implementation-code changes.

9.13.3 Audit

All change units with the `[test-spec-change]` tag are aggregated into a substrate-native audit-feed for the architect. The aim is to track a pattern: if the AI agent frequently requests a criteria change, this is a signal of a problem with the formulation of the source requirement (chapter 7 ADAPT, the backward-findings categories `terminology` or `gap`).

9.13.4 Judge isolation (P7) — a special case of protection

`judge.vendor` + `judge.model` mandatorily differs from the production model of the verifiable SPEC-AI (§9.6.2). A coincidence is blocked by a substrate hook.

9.14 Engineer's spot-check

9.14.1 Normative procedure

Once per iteration (by default — the regular implementation cycle; the specific interval is fixed in the project conformance manifest) the engineer performs a spot-check of 5 random TC in status `passing`. The aim is to catch the situation where the AI agent generated a "green" TC that verifies nothing meaningful (an `assert True` equivalent; a VLM-prompt that passes on a blank screen; an eval criterion that always returns pass on the baseline).

9.14.2 Sampling

The sample is **directed by machine signals** rather than drawn at random. Once red history is in place (§9.18.2), blind sampling is redundant: the substrate knows which tests are suspect.

Parameter	Normative requirement

Sample size	5 TC (by default); MAY be increased in the project conformance manifest
Selection priority	1) TC **without red history** (the fixing run was never observed red); 2) TC that **survived a mutant** (mutation checking killed no mutant), where the substrate performs it; 3) TC of the `implementation-originated` class (§9.18.2); 4) the rest — topped up at random to the sample size
Distribution across types	Uniform across `tc-type` within the random top-up
Status	Only `passing`
Who selects	The AI agent, by the priority above; the random top-up uses substrate-native randomness (the seed is fixed in the audit-feed)

Random sampling catches a weak test with a probability inversely proportional to the size of the set; directed sampling puts in front of the engineer exactly those tests for which **the machine has already found a sign of toothlessness**. Cheaper and more precise.

9.14.3 What the engineer checks

1. Does the Pass / Fail criterion match the stated behavior in the verifiable artifact?
2. Is the VLM-judge or eval-judge criterion too lenient?
3. Are the preconditions real (not substituted via a seed that masks a bug)?
4. Does the Out-of-scope cover exactly what the paired TC is supposed to cover?

9.14.4 Recording the result

The spot-check result is recorded substrate-natively:

```
last-spot-check:
  date: "<ISO-date>"
  by: "<engineer-id>"
  sampled-tests: [TC-NN, TC-NN, ...]
  issues-found: integer
  issues:
    - test: "TC-NN"
      issue: "<short description>"
```

9.14.5 Reaction to findings

On `issues-found > 0`:

- The architect registers a change unit to fix the found TC.
- The AI agent MUST account for the identified pattern in subsequent TC generations; the pattern is added to the agent's system prompt or to the meta-style guide.
- On a repeated occurrence of the same pattern — escalation to a review of the TC generation template.

9.15 Coverage matrix (auto-generated)

9.15.1 COVERAGE artifact

`COVERAGE.md` (the substrate-native artifact name) is an auto-generated summary report of requirements and specifications coverage by test cases at the level of the requirements substrate. It is marked with a substrate-native auto-generated flag.

9.15.2 Mandatory metrics

Metric	Goal	Action on violation
`coverage-percent` (verified / total artifacts)	The target threshold is fixed in the conformance manifest	A substrate-native gate blocks promotion
`approved` without `verified`	0 before promotion	AI-agent backlog for the next iteration
Coverage by a paired negative TC	100% of statements	The AI agent creates a change unit with a paired negative
`passing-tests / total-tests`	100% before change-unit promotion	Blocks QG-2 (§9.10)
`manual-pending` overdue	0	Notification to the architect; blocking of the affected artifacts
Stale (`last-run.requirement-version` < current)	0	The AI agent re-runs the run

The two coverage metrics are counted in **different units**, and substituting one for the other is prohibited. `coverage-percent` measures **artifact promotion** (how many artifacts reached `verified`) — the unit of count is the artifact. The pos/neg pairing rule (§9.7, MVR-5) measures **verification completeness** — the unit of count is the normative assertion, and the target here is always 100%, with no threshold in the manifest. An artifact with a green `coverage-percent` may still contain an assertion covered only by a positive TC; this is precisely why QG-2 blocks the promotion per assertion (§10.5.2) rather than on the presence of a single negative TC per artifact.

9.15.3 Regeneration triggers

`COVERAGE.md` is regenerated automatically on:

- Completion of a change unit with a change to requirement / SPEC / TC artifacts;
- Promotion of a change unit into the substrate main line;
- Every successful runner run (an update to `last-run`);
- On a schedule (a substrate-native scheduler).

9.16 Delta-TZ and TC

9.16.1 Impact analysis over tests

On a delta-TZ (chapter 7 §7.6) the AI agent performs an impact analysis over TC simultaneously with the impact analysis over requirements:

1. Finds all TC whose `verifies[].requirement-version` is below the new version of the verifiable artifact.
2. Marks them `obsolete-pending: true`.
3. Forms a table of affected TC in the frontmatter of the delta-ADAPT or the associated change unit:

TC	Verifies	Old version	New version	Action
TC-NN	SR-NN	v1.1	v1.2	Update (new step)
TC-NN	SR-NN	v1.1	v1.2	No change (still current)
TC-NN	BR-NN	v1.0	deprecated	Move to `obsolete`

4. Generates updated versions of the TC in the same change unit as the delta-ADAPT.
5. After running the updated TC and their transition into `passing` — removes `obsolete-pending` and updates `verifies[].requirement-version` to the new artifact version.

9.16.2 TC transition into `obsolete`

A TC moves into `obsolete` (terminally) if:

- The parent artifact (BR / SR / SPEC) was moved into `deprecated` without a replacement;
- The parent artifact was replaced by a new one (`replaced-by`) for which a new set of TC was created, **and** the old set does not cover the behavior of the new artifact;
- The behavior covered by the TC no longer exists in the new version of the artifact.

An `obsolete` TC is immutable and is not deleted (V1; see chapter 3 §3.3.1).

9.17 Storage layout

Test cases are stored in the `tests/` subfolder of the requirements substrate. The substrate-native storage implementation is substrate-specific (see guide/03 for distributed VCS; guide/04 for a document-oriented store).

9.17.1 At the system / subsystem level

```
[requirements-substrate]/      # system or subsystem scope (chapter 6 §6.11)
  br/   sr/   tr/                # chapter 6
  specs/                          # chapter 8
  adapt/                          # chapter 7
  tests/
    business/    TC-NN-*.md
```

```
    system/      TC-NN-*.md
    ux/          TC-NN-*.md
    contract/    TC-NN-*.md
    eval/        TC-NN-*.md
    security/    TC-NN-*.md
    baselines/                    # for ux / eval
      <baseline-artifact>.png
      <eval-dataset>.jsonl
  COVERAGE.md                     # auto-generated (see §9.15)
```

9.17.2 Implementation in the code substrate

The TC implementation (code) lives in the code substrate, separately from the requirements substrate; it is addressed by the `automation.location` field (a substrate-native pointer). The TC↔implementation relationship: 1:1.

9.18 Test authorship isolation and red history

9.18.1 The three axes of isolation (P8)

A test created in the course of writing the code will be tuned by the agent **to the code**, not to the requirement: it will honestly verify what was written — but not what was required. The existing protection (§9.13) forbids changing the code and the criteria of an **existing** TC within one atomic change unit, but it does not forbid writing a **new** TC after the code. Isolation closes this along three axes.

Axis	Normative requirement	Check
Time	The TC is frozen (`ready`) **before** the implementation it verifies is started. A TR MUST NOT enter work while the TC linked to it are not in `ready`	A substrate hook blocks the start of the TR
Author	The agent creating the TC **does not coincide** with the agent creating the implementation (different sessions or models). An extension of P7 from `ux` / `eval` to all TC types	The TC provenance (`ai-provenance.generated-by`) is cross-checked against the provenance of the implementation change unit
Change	An edit to the Pass/Fail criteria of a frozen test — only through `[test-spec-change]` with an engineer's approval (§9.13.2)	A substrate hook

9.18.2 Red history (P9)

> ***A test that has never been observed red is not evidence.***

The **fixing run** is performed **before** the implementation: the test MUST be **red** — the functionality under verification does not exist yet. The "red → green" transition is recorded by the substrate as part of the run history and is a **precondition** for the TC to count as evidence at QG-2.

A green result on the fixing run is a **signal to investigate**, not a success: either the test verifies nothing, or the functionality under verification already exists. Both cases require investigation by the engineer, and neither permits the test to be counted as evidence.

Inheritance. For tasks that do not change observable behavior (refactoring, rebuild, migration without a contract change), red history is **inherited** from the TC of the same statement: demanding a fresh red run there is pointless — the behavior has not changed.

Exception — tests of the `implementation-originated` class (§6.13). Such a test is written **after** the code and is born green: by construction it cannot have a red history. Compensation is mandatory: for this class of TC, **mutation checking is mandatory** — a test born green proves that it works by a **killed mutant**. Without a killed mutant such a TC is not evidence.

9.18.3 The machine signal of a weakened norm

> *If, after an edit to a test, **a previously green system fails**, then the norm has changed **in fact**, however the edit was classified.*

The signal is machine-verifiable and does not depend on whether the agent honestly tagged the change unit `[test-spec-change]`. Firing is blocking: the change unit does not pass until the edit is carried through as a change to the norm (with an engineer's approval) or reverted.

This is the paired protection to P8: isolation closes off tuning the test to the code; the machine signal closes off the silent weakening of a test to fit code that is already written.

9.19 AT — the acceptance test of the contractual contour

9.19.1 Purpose

TC traceability is closed through **interpretation**: `TC → SR → ADAPT → TZ`. Hence a class of defects that the TC level cannot catch **in principle**: if the interpretation of the TZ is wrong, all TC may be `passing` — the system conforms perfectly to the **wrong** interpretation — and still fail acceptance at the client.

AT (`AT-NN`) is an acceptance test derived **exclusively** from the **effective TZ** (§7.14): the initial TZ with its annexes plus every signed ACTZ. An AT verifies conformance to the **contract**, not to the interpretation.

9.19.2 Isolation as the generation mechanism

An AT is created by an **isolated agent**: the input is **only** the effective TZ. Access to ADAPT, BR / SR / SPEC, TC, and the code is **prohibited**. The model of the AT-generating agent MUST differ from the model of the main agent (mirroring P7).

Isolation here is not hygiene but the **essence of the mechanism**: an AT is a simulation of the client's view. An agent that has seen the ADAPT or the SR will reproduce **the same** interpretation — and the AT will start confirming that interpretation instead of the contract. The acceptance level would collapse into the verification level, and an error of construal would receive a **false confirmation of conformance**. A breach of isolation is **fatal** (§10.11.1).

9.19.3 Mandatory fields and body

Field	Obligation	Value
`id`	mandatory	`AT-NN`; immutable
`verifies[]`	mandatory	Only `TZ §N` and `ACTZ-NNN §M`. A reference to an internal artifact (BR / SR / SPEC / TC) is **fatal**
`tz-version`	mandatory	The revision of the effective TZ from which the AT was derived (§9.19.4)
`generator`	mandatory	The provenance of the isolated agent: vendor, model, confirmation that it had no access to the internal contour
`negative`	mandatory	Pos/neg pairing — as for a TC (§9.7)
`status`	mandatory	`draft → ready → passing / failing → obsolete` — the same state machine as a TC (§9.9)
`environment-ref`	mandatory	The bench and dataset of the acceptance trials (§8.5.10). **The field is not filled in by the generator**: the isolated agent does not see the internal artifacts and MUST NOT see them (§9.19.2). The binding to the bench is set by the architect or the runner **after** generation — so that the trials remain reproducible and the isolation is not broken
`automation`	mandatory	As for a TC, including the mandatory `automation.kind` (§9.3): the run is performed only by an automated runner
`last-run`	runner-managed	As for a TC (§9.12)

`tz_text` — the verbatim quotation of the effective-TZ clause under verification, placed next to the verification steps — is a **mandatory** body section of an AT. The quotation settles the argument at acceptance: what is produced is not a paraphrase but the clause of the contract itself.

9.19.4 Regeneration before the trials

> ***AT are regenerated before every round of trials*** *from the* ***current*** *revision of the effective TZ. The trials programme MUST correspond to the revision of the contract against which the system is presented; an outdated programme* ***MUST NOT be admitted*** *to the trials.*

The effective TZ lives incrementally: each signed ACTZ yields a new revision. AT derived at planning time are stale by the time of the trials — and without regeneration, at the end of a long engagement the system is checked against a year-old contract. Regeneration is cheap: the same isolated agent performs it.

The freshness of the AT relative to the current revision of the effective TZ is checked by a gate; a divergence blocks the trials.

9.19.5 Failure routing

A divergence between the levels is **diagnostics**, not noise:

AT	TC	Diagnosis	What gets fixed
✗	✓	**Interpretation error**: the system conforms to the ADAPT but not to the contract	The ADAPT / ACTZ, then the derived requirements

✗	✗	Implementation defect	The code
✓	✗	The requirement is stricter than the contract, or the TC is wrong	Investigation: a redundant requirement or a defective TC
✓	✓	Normal	—

The first row is what the AT is introduced for: this defect is detectable by no TC.

9.19.6 The acceptance report

`ACCEPTANCE.md` is an auto-generated report (the analogue of `COVERAGE.md`, §9.15): AT coverage across the sections of the effective TZ and the clauses of the signed ACTZ. The product is not presented for delivery until all AT are in status `passing` (§10.4.3).

9.19.7 Binding a test to a task (TR)

The acceptance criteria of a TR are already normalized (§6.7.2, §6.7.3) and are checked by QG-2 (§10.6.2); a TR does not introduce an artifact class of its own. The gap lies elsewhere: the criteria of a TR are verified by the tests of the parent SR, **which are broader than the scope of the task**, and no local check of "has exactly this task been done" exists.

The following TC fields are introduced:

Field	Value
`verifies-tr`	The task (TR) to whose scope the test is bound
`verifies-claims[]`	The **subset** of the parent SR's normative statements covered by the test within the scope of this task

The right to narrow is the essence of the mechanism. A TR-bound test MUST be entitled to verify not the whole SR but exactly those of its statements that fall within the scope of the task. Without that entitlement the field is useless: the test again turns out broader than the task, and locality is not achieved.

The binding **does not weaken** coverage: completeness is still counted from the statements of the artifact (§9.7), not from the tasks. `verifies-tr` yields a *local* readiness criterion for a TR, not an alternative way to close an SR.

9.20 Relation to other chapters

Chapter	Relation
02 Positioning in the methodology typology	TC — the bottom layer of the trace chain (Statement 1, Source-of-Truth inversion); pinning the requirement through `verifies[].requirement-version` (Statement 3, substrate versioning)
06 Requirements hierarchy	TC verifies BR / SR; `verified-by[]` — an auto-derived inverse edge on the requirement side
07 ADAPT	TC → SR → ADAPT → TZ — the full trace chain; backward findings (`terminology`, `gap`) are fed by patterns from the `[test-spec-change]` audit (§9.13.3)

08 Specifications	Spec-specific TC per the table in §9.8; SPEC → `verified-by[]` auto-derived; type-specific extensions §9.6
10 Lifecycle and QG	the TC state machine; the QG-0 + QG-1 bundle for TC (`draft → ready`); QG-2 for the verifiable artifact requires pos/neg pairing and spec-specific TC kinds
03 Substrate versioning	Immutable IDs (V1); atomic change unit and hooks (V2 + V3); diff & review for `[test-spec-change]` (V3); TC versioning without loss of history (V4); pinning `verifies[].requirement-version` (V5); author + timestamp for `last-run` (V6)
11 Maturity model	RENAR-1: TC mandatory; RENAR-3+: pos/neg pairing 100%, spec-specific TC table mandatory; RENAR-4+: AI-generated + AI-executed
13 Conformance	Closed list of TC types (§9.5) — a mandatory clause of v1.0; spec-specific TC table (§9.8) — a mandatory clause of v1.0; pos/neg pairing — a mandatory clause of v1.0; judge ≠ production isolation — a mandatory clause of v1.0
reference/02 — schemas	The full machine-readable schema of the TC frontmatter + type-specific extensions

10. Lifecycle and Quality Gates

> ***Part of the RENAR Standard v1.0*** · *← Table of contents*
>
> ***Dense chapter:*** *read after §6–§9; passing the QGs — guide/00; density — reference/09.*

10.1 How artifacts move: statuses and gates

A RENAR artifact — a requirement, a specification, an ADAPT, a test — does not sit still. It moves through states: `draft → approved → verified → …`. And it cannot move arbitrarily: every transition is guarded by a **Quality Gate** — a condition that MUST be satisfied, or the transition does not happen. A requirement does not become `verified` until all of its tests have gone green on the current version; an ADAPT does not become `approved` until every finding is closed by a signed clarification protocol. A gate is not a "success checkmark" but a check that is entitled to say "no" and leave the artifact where it is.

This chapter brings together the state machines of all artifacts and normalizes the gates: what is checked before each transition, who MUST check it and when. The chapter fixes **only** states, transitions, and gates; artifact frontmatter is defined by chapters 6–09.

10.1.1 Decision tree: which gate now (informative)

```
flowchart TD
  S[Artifact change] --> T{Transition type?}
  T -->|draft → approved| Q0[QG-0 approval]
  T -->|draft → ready| Q1[QG-1 implementation]
  T -->|ready → verified| Q2[QG-2 verification]
  T -->|architecture sign-off| Q3[QG-3 optional]
  T -->|client accept| Q4[QG-4 optional]
  Q0 --> V0{preconditions §10.3.1}
  Q1 --> V1{TC automated §10.3.2}
  Q2 --> V2{passing-tests §10.3.3}
  V0 -->|fail| X[Stays draft]
  V1 -->|fail| X1[Stays approved]
  V2 -->|fail| X2[TC failing]
```

10.2 Normative definition of a Quality Gate

10.2.1 Quality Gate

Quality Gate (gate) is a normative condition whose check MUST be performed for a permitted transition of an artifact from one lifecycle state to another. Each gate consists of:

1. **Identifier** — `QG-N` or `QG-<artifact>-<state>` (closed list §10.3, §10.4).

2. **Precondition** — a set of checkable assertions about the artifact and related artifacts that MUST be true at the moment the gate is triggered.
3. **Postcondition** — the state the artifact transitions into after a successful gate pass, and the observable effects (for example, the appearance of a record in the transition log §10.13).
4. **Trigger** — who or what initiates the gate check (participant: AI agent / Architect / automated runner; event: approval / run completion / arrival of a delta-TZ).
5. **Control point** — the place in the substrate where the check MUST be automated (§10.11).

A gate is not a success event — it is a condition that **MUST be checked**. A gate pass MAY be negative (the precondition is not satisfied) — in that case the transition is prohibited and the artifact stays in its current state.

10.2.2 Who MUST check a gate

Gate type	Required participant	Substrate enforcement
Approval (QG-0)	Architect or authorized role-holder on the substrate	Atomic recording of authorship and time (V6, §3.3.6)
Implementation (QG-1)	Automated runner (CI, eval-runner)	Atomic recording of the run result pinned to the artifact version (V5, §3.3.5)
Verification (QG-2)	Automated runner with `version-pin` confirmation	V5 + V6
Architecture (QG-3, optional)	Architect's signature (ADAPT / SPEC-ARCH)	V3 + V6
ACTZ signing	Dual signature (client + vendor)	V3 + V6
Acceptance (QG-4, optional)	Stakeholder with authority	V6

10.2.3 Relationship to SENAR

SENAR §8 describes Quality Gates as an abstract concept for AI-driven development. RENAR **extends** SENAR in the requirements-engineering domain:

- It keeps the identifiers QG-0 / QG-1 / QG-2 as mandatory.
- It normalizes **formal state machines** for each artifact type (SENAR does not do this).
- It binds every transition in a state machine to a concrete gate with preconditions and postconditions.
- It adds optional QG-3 / QG-4 for industries with extended audit requirements.

RENAR does not contradict SENAR; an implementation is SENAR-compatible with RENAR if the requirements of §10.3 + §10.11 are met.

10.3 Canonical RENAR gates (mandatory)

A closed list of three mandatory gates. Extensions outside this list are only the optional ones in §10.4 or through the formal standard change procedure §10.10.

10.3.1 QG-0 — approval gate

Purpose: permits an artifact to transition from draft into a state approved for development.

Precondition (common part, supplemented per-artifact in §10.5–§10.9):

- The artifact frontmatter is valid against the schema of its chapter.
- The artifact identifier is unique in the substrate (V1, §3.3.1).
- An adversarial review has been performed; or its non-applicability is explicitly recorded — permitted **only** for trivial artifacts (by the criteria declared in the conformance manifest, §13) with the reason recorded in the transition log (§10.13).
- If the artifact references a source (`source.adapt` for BR/SR/SPEC, `verifies[]` for TC) — the referenced artifact exists in the substrate in a state no lower than `approved` .

Postcondition:

- The artifact transitions into `approved` (for requirements / SPEC) or `ready` (for TC) or `approved` ADAPT (§10.8).
- A record in the transition log (§10.13).
- For requirements / SPEC: decomposition into child artifacts is permitted (for BR — SR; for SR — TR + SPEC via `constrained-by` / `implements-spec`).

Trigger: explicit approval by the Architect / role-holder through the substrate's native mechanism (V3 diff & review, §3.3.3).

Applicable artifacts: BR, SR, TR, SPEC, ADAPT, TC.

10.3.2 QG-1 — implementation gate (TC only)

Purpose: confirms that a valid implementation exists for the artifact — code, configuration, an infrastructure artifact — suitable for verification.

Precondition:

- The implementation is pinned to the artifact version via `version-pin` (V5, §3.3.5).
- `automation.status: automated` (with a valid `automation.location`) or `automation.status: manual-pending` (with `manual-pending-until` set and `manual-pending-reason`).
- All static checks of the implementation by the substrate agent (types, lint, schema) — passed.
- Pos/neg pairing for the artifact's covered assertions is ensured (chapter 9 §9.7).
- All mandatory body sections of the TC (chapter 9 §9.4) are filled in.

Postcondition:

- The TC transitions into `ready` .
- A record in the transition log.

Trigger: the approving participant (one-click promote `draft → ready`) upon the automated runner's confirmation of a passing dry-run.

Applicable artifacts: TC (`draft → ready`).

Note: TR does not pass a separate QG-1 gate. The implementation-validity conditions (impl scope, version-pin, static checks) for TR are part of the QG-2 preconditions (§10.6.2). **QG-1 applies only to TC.**

For BR / SR / SPEC the `approved → verified` transition is governed by a single QG-2; there is no intermediate QG-1 implementation gate for requirements and SPEC.

10.3.3 QG-2 — verification gate

Purpose: confirms that the system's observed behavior conforms to the artifact: all TCs in the artifact's `verified-by` are in state `passing` on the artifact's current version.

Precondition:

- For BR / SR / SPEC: all TCs in `verified-by` have `last-run.result = pass`, and `last-run.requirement-version` (or the equivalent `spec-version` / `version`) matches the current `version` of the verified artifact.
- Pos/neg pairing on the artifact's normative assertions — satisfied.
- **Red history** (§9.18.2): every TC in `verified-by` has a recorded "red → green" transition. A TC that was never observed red is not evidence and does not close QG-2. The exception is a TC of the `implementation-originated` class (§6.13): for it a **killed mutant** is mandatory.
- All mandatory spec-specific TC kinds for the artifact type are present (chapter 9 §9.8).
- For TR: all its AC are verified by bound TCs (`last-run.result = pass`).
- Spec-specific additional preconditions:
  - SPEC-UI / SPEC-AI: TC in state `passing` with `judge-isolation` observed (chapter 9 §9.13.4).
  - SPEC-SEC: a TC `tc-type: security` is present and `passing`.

Postcondition:

- The artifact transitions into `verified`.
- A record in the transition log with evidence-refs (a list of run IDs).
- The substrate MUST record the `version` of the verified artifact in the evidence record (V5).

Trigger: the automated runner confirms passing TCs and initiates the promote-transition on the author's request (one-click promote `approved → verified`).

Applicable artifacts: BR, SR, SPEC, TR.

10.4 Gates outside the mandatory core

This section describes two different things, and they must not be conflated.

The optional QG-3 and QG-4 (§10.4.1, §10.4.2) are normatively described but **not mandatory** for conformance (chapter 13). An implementation MAY declare in the conformance manifest either support for QG-3 / QG-4 or their absence. Conformance without QG-3 / QG-4 remains valid.

The release acceptance gate (AT) (§10.4.3) is **mandatory** and is not optional. It does **not** belong to the closed list of Quality Gates (§10.10.1) and does not extend it: a QG normalizes the transition of an **artifact** between states, whereas the AT release gate normalizes producing the **product** for trials and has no `QG-N` of its own.

10.4.1 QG-3 — architecture gate (optional)

Purpose: permits an ADAPT to transition from `answered` to `approved` (§10.8). Also applicable to SPEC-ARCH decomposition decisions in projects with regulated architectural acceptance.

Precondition:

- All backward findings in the ADAPT are in status `resolved` (chapter 7 §7.4.5).
- Every backward finding carries a `decided-in` pointing to a clause of a **signed** ACTZ (chapter 7 §7.13).
- The Architect's signature is ready (chapter 7 §7.5). A client signature on the ADAPT is **not required**: the client's obligations are carried by the ACTZ.
- For SPEC-ARCH (if QG-3 is applied): the decomposition decision is recorded in the substrate as an ADR-like artifact with a reference from the SPEC-ARCH (the ADR form is substrate-specific — it belongs in `guide/`).

Postcondition:

- The ADAPT transitions into `approved` (immutable, on a par with the TZ).
- A record in the transition log with the Architect's signature (V6 author + timestamp).

Trigger: the Architect's signature on the ADAPT (§7.5). The dual signature stands on the ACTZ (§7.13) and is recorded atomically (V2 atomic change unit, §3.3.2).

When to apply:

- ADAPT — always (but an implementation MAY declare QG-3 as a local alias for ADAPT approval, without carving it out as a separate gate).
- SPEC-ARCH — in projects with regulatory requirements for architectural acceptance.

10.4.2 QG-4 — acceptance gate (optional)

Purpose: records the client's acceptance of the business outcome after release. The transition of a BR from `verified` to `accepted`.

Precondition:

- BR in `verified` (QG-2 passed).
- The measurable business outcome (`business-outcome` in the BR frontmatter) — measured; `current-value` recorded.
- `achievement` ≥ the project-configurable threshold (80% by default, fixed in the conformance manifest).
- A formal Stakeholder signature.

Postcondition:

- The BR transitions into `accepted` (a terminal non-degradable status — a reverse transition requires a delta-TZ).
- A record in the transition log with the Stakeholder signature.

Trigger: formal acceptance by the Stakeholder upon release.

When to apply:

- Projects with explicit recording of post-release outcomes (product SaaS, regulated industries).

- In the absence of QG-4 — the `accepted` status is not used; the BR stays in `verified` until `deprecated`.

10.4.3 The release acceptance gate (AT)

Purpose: the product is not submitted for acceptance trials until its conformance to the **contract** is proven.

Precondition:

- All AT (§9.19) are in status `passing`.
- The AT are derived from the **current** revision of the effective TZ (§7.14): the `tz-version` of every AT matches the current revision. An outdated trials program is **not admitted** to the trials (§9.19.4).
- The provenance of the AT generator confirms the isolation: the model differs from that of the main agent, and there was no access to the internal contour (§9.19.2).

Postcondition: the product MAY be submitted for trials; the `ACCEPTANCE.md` report is current.

Trigger: the isolated AT-generating agent + the automated runner.

The gate is **mandatory** and is not conflated with QG-4 (§10.4.2): QG-4 is optional and measures the business outcome, whereas the AT release gate measures conformance to the contract. A business goal MAY be achieved while the contract is not conformed to, and vice versa.

10.4.4 Conformance with optional gates

The conformance manifest (chapter 13) MUST explicitly declare:

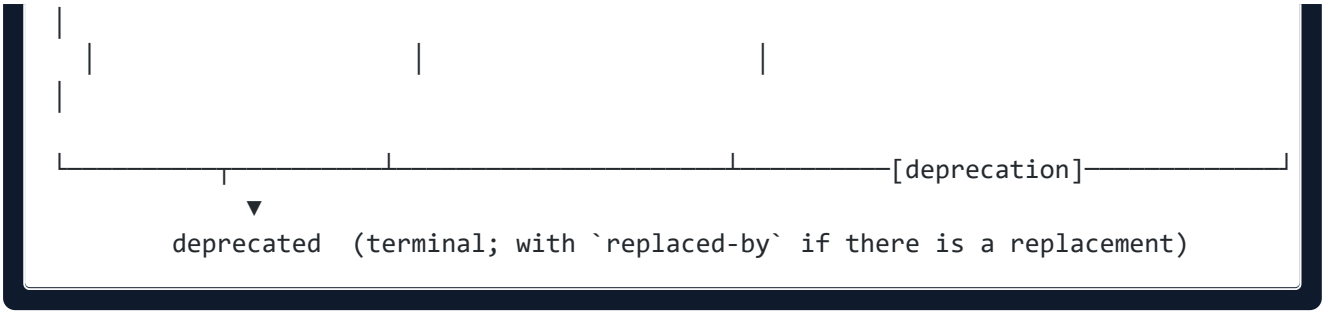
```
quality-gates:
  qg-0: required          # always required
  qg-1: required
  qg-2: required
  qg-3: declared          # required | declared | absent
  qg-4: declared
```

`declared` means: the implementation supports the gate; artifacts MAY pass it, but conformance does not require passing it for all artifacts. `absent` — the gate is not applied in the implementation; the artifact's terminal state is `verified` (without `accepted`).

10.5 BR / SR state machine

10.5.1 States and transitions

```
draft  ──[QG-0]──▶  approved  ──[QG-2]──▶  verified  ──[QG-4 (optional)]──▶
accepted
  │                    │                     │
```



Status	Semantics	Transition gate
`draft`	Created by an AI agent or the Architect; not yet approved	— (creation)
`approved`	Approved for decomposition / implementation	QG-0 (§10.3.1)
`verified`	All derived TCs `passing` on the current version	QG-2 (§10.3.3)
`accepted`	Post-release business outcome confirmed	QG-4 (§10.4.2, optional)
`deprecated`	Terminal; not deleted (V1 immutable history)	Deprecation transition (§10.5.3)

BR / SR frontmatter (including the mandatory status fields) is defined in chapter 6 §6.5.2 / §6.6.2. This section normalizes **only** the transitions between states and the binding to gates.

10.5.2 Per-transition preconditions

Transition	Gate	Additional preconditions (on top of §10.3)
`draft → approved`	QG-0	BR: `business-outcome` filled in. SR: the `parent` BR in a state no lower than `approved`. If the SR references a SPEC via `constrained-by[]` — all SPECs in `approved` or higher.
`approved → verified`	QG-2	Pos/neg pairing **per normative assertion** of the artifact (§9.7, MVR-5): every assertion describing observable behavior is covered by a positive + negative TC pair. The sole exception is an assertion that itself states a negative invariant. All `last-run.requirement-version` match the current `version`.
`verified → accepted`	QG-4	Only if the implementation declared QG-4.
`* → deprecated`	Deprecation transition (§10.5.3)	See §10.5.3

10.5.3 Deprecation transition

Precondition:

- The artifact is in any non-`deprecated` state.
- `replaced-by` (if specified) exists in the substrate and is in a state no lower than `approved`.
- There are no active child TRs in state `approved` and no active implementer tasks on this artifact (atomic re-pointing of tasks to the replacement is a mandatory condition, V2 atomic change unit).

Postcondition:

- `status: deprecated`.
- `deprecated-date` recorded (V6).
- `replaced-by` specified, if there is a replacement.

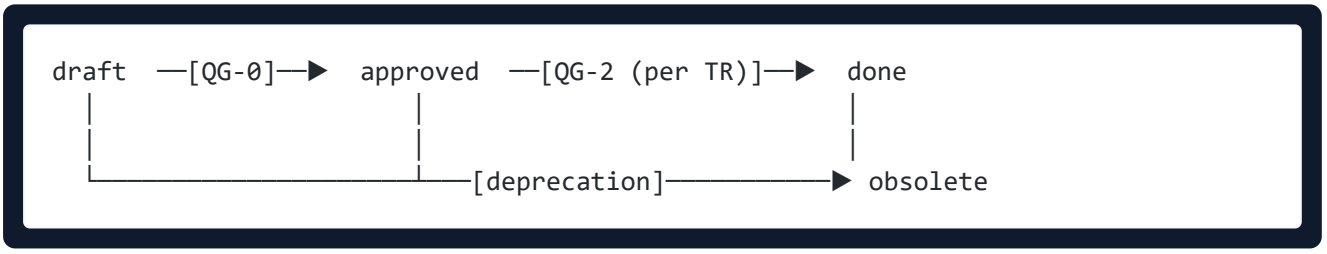
Trigger: the Architect or the Product Owner.

10.5.4 Reverse evolution of verification

If an artifact is already `verified` but its `version` has been incremented (for example, after applying a delta-ADAPT, chapter 7 §7.6) — the status MUST revert to `approved` before re-passing QG-2 on the new version. This transition is mandatory and automatic: the substrate MUST invalidate `verified` upon a change of `version` (chapter 6 §6.5.4 reverse evolution).

10.6 TR state machine

10.6.1 States and transitions



Status	Semantics	Gate
`draft`	TR created; AC not yet finalized	—
`approved`	AC approved; implementer work may start	QG-0 (§10.3.1) with the TR preconditions from §10.6.2
`done`	AC verified; all bound TCs `passing`	QG-2 (§10.3.3) with the TR preconditions from §10.6.2
`obsolete`	TR has lost relevance before completion (the parent SR changed)	Deprecation (Architect)

10.6.2 Preconditions for TR

QG-0 for TR (`draft → approved`) — in addition to the common part:

- A goal is stated (`goal`).
- AC are verifiable and independent (each AC is a separate check).
- At least one negative scenario is present.
- A reference to the parent SR (or BR for simple configurations) is established via `implements`.
- If the TR implements a SPEC — the mandatory field `implements-spec[]` (chapter 8 §8.6.2).
- If the task touches security — `threat-surface` is declared (chapter 8 §8.5.8).

QG-2 for TR (`approved → done`):

- All TR AC are confirmed by the passing of corresponding TCs (`last-run.result = pass`).
- Pos/neg pairing for each AC.
- For a TR implementing a SPEC: a TC of the corresponding spec-specific kind exists and is `passing` (chapter 9 §9.8).

10.6.3 TR deprecation

If the parent SR transitions into `deprecated` or its `version` has changed such that the TR's AC are no longer relevant — the TR transitions into `obsolete` . This is **not** degradation — it is an alternative terminal path. A TR in `obsolete` is not deleted.

10.7 SPEC state machine

10.7.1 States and transitions

Status	Semantics	Gate
`draft`	SPEC created; mandatory frontmatter fields being filled in	—
`review`	Mandatory body sections (§8.4.1) and type-specific ones (§8.5) present; ready for review	Review-transition (§10.7.2)
`approved`	Architect confirmed; `depends-on[]` consistent	QG-0
`verified`	All mandatory spec-specific TCs `passing`	QG-2
`obsolete`	Replaced or no longer relevant; `replaced-by` mandatory	Deprecation (§10.5.3 mutatis mutandis)

10.7.2 Review-transition (`draft → review`)

The review-transition is not a full-fledged gate in the sense of §10.2.1 — it is an automatic check of structural completeness. **Precondition**:

- All mandatory frontmatter fields §8.4 are filled in.
- All mandatory body sections §8.4.1 are present.
- Type-specific sections §8.5 are present for the corresponding `spec-type` .

Postcondition: `status: review`; the artifact is visible to the Architect for review.

If the review-transition is not passed — the artifact stays in `draft`; the substrate MUST return the list of missing sections (V3 diff & review supports structural feedback).

10.7.3 Preconditions for SPEC

QG-0 for SPEC (`review → approved`) — in addition to the common part §10.3.1:

- The `depends-on[]` graph is acyclic (chapter 8 §8.6.3).
- All SPECs in `depends-on[]` are in a state no lower than `approved`.
- If the SPEC references an ADAPT via `source.adapt` — the ADAPT is in state `approved`.

QG-2 for SPEC (`approved → verified`):

- All mandatory spec-specific TC kinds for the `spec-type` are present and `passing` (chapter 9 §9.8).
- For SPEC-AI: pos/neg pair coverage over `evaluation-criteria` = 100%; judge-isolation observed.
- For SPEC-SEC: `tc-type: security` is present.
- For SPEC-DATA: `tc-type: contract` is present for published interface fields.

10.8 ADAPT state machine

10.8.1 Macro-states

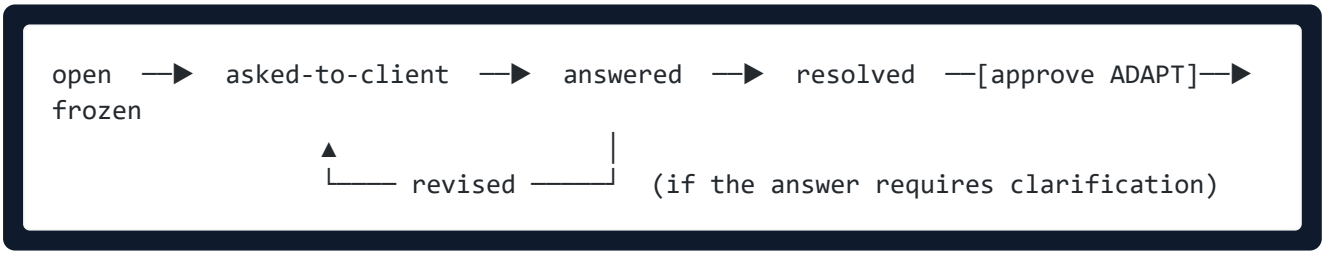
The ADAPT states (`draft → review → asked → answered → approved → frozen`, and the terminal `superseded` upon supersession) are defined in chapter 7 §7.4 and §7.6.4. The state

`client-ready` is withdrawn: putting questions to the client is the subject of an ACTZ (§7.13), not a state of the ADAPT. This section normalizes the gates.

Transition	Gate	Precondition
`draft → review`	Review-transition	The Forward interpretation covers all TZ sections; the primary backward findings are recorded in `open`
`review → asked`	Backward-ready	All backward findings moved to `asked-to-client`; the questions are put to the client as part of an ACTZ (`status: sent`)
`asked → answered`	Client-return	All backward findings in `answered`; the decisions are recorded as clauses of a **signed** ACTZ (`status: signed`, dual signature V6)
`answered → approved`	**QG-3** (§10.4.1)	All backward findings in `resolved` and carry a `decided-in` pointing to a clause of the signed ACTZ; the Architect's signature is ready
`approved → frozen`	Freeze-transition	Automatic after approve; the ADAPT is immutable; generation of BR / SR / SPEC with `source.adapt = approved` is permitted
`frozen → superseded`	**QG-3 of the superseding ADAPT** (§10.8.5)	The superseding ADAPT (`supersedes: ADAPT-MMM`) has reached `approved`; derived BR / SR / SPEC are re-pointed or re-derived

10.8.2 Nested state machine for a backward-finding record

Each backward-finding record inside an ADAPT has its own subordinate state machine (chapter 7 §7.4.5):



Sub-state	Semantics
`open`	Recorded by the Engineer; not sent to the client
`asked-to-client`	Sent to the client; the question date is recorded
`answered`	The client answered; the answer recorded (V6 author + timestamp)
`resolved`	The Engineer integrated the answer into the Forward interpretation
`revised`	The answer is vague; a repeat question (return to `asked-to-client`)
`frozen`	After ADAPT approval; changes are impossible

Normative rule: QG-3 (approve ADAPT) is **prohibited** if at least one backward-finding record is in `open` / `asked-to-client` / `answered` / `revised`. All such records MUST be in

`resolved` (chapter 7 §7.4.5).

10.8.3 QG-3 for ADAPT — detailed

Precondition (full):

- All Forward-interpretation sections are filled in (the forward-complete criterion).
- All backward-finding records in `resolved`.
- Every backward finding carries a `decided-in` pointing to a clause of an ACTZ in status `signed` (chapter 7 §7.13).
- The Architect's signature obtained and recorded by the substrate's native mechanism (V3 + V6).
- If the ADAPT is a delta-ADAPT: the parent-ADAPT in `frozen` (chapter 7 §7.6).

Postcondition:

- The ADAPT transitions into `approved`.
- A record in the transition log with the Architect's signature.
- An ADAPT with at least one finding not closed by a signed ACTZ is **not** transitioned into `approved`: an interpretation MUST NOT rest on a decision the client never took.

Trigger: explicit approval by the Architect.

10.8.4 Errata for a frozen ADAPT

`frozen` is a terminal state along the derivation line. Changes are possible only by adding a new artifact via one of three paths:

1. **Delta-ADAPT** (if the TZ contains an ambiguity discovered late) — a new artifact with an explicit `parent-adapt` link.
2. **Errata-ADAPT** (if there is an interpretation error by the engineer) — a separate artifact. If the correction touches a decision of a signed ACTZ (it changes an obligation), a **new ACTZ** with the dual signature is REQUIRED; if it does not, the Architect's signature suffices.
3. **Supersession** (if the prior decision was correct but later refuted) — a superseding ADAPT transitions the superseded one into `superseded` (§10.8.5, chapter 7 §7.6.4).

In all three cases the frozen ADAPT is **not edited**. This is a V1 requirement (immutable history) for contractual artifacts (chapter 7 §7.6.3).

10.8.5 The frozen → superseded transition (supersession)

Supersession of an approved/frozen ADAPT is normalized in chapter 7 §7.6.4. The lifecycle transition:

```
frozen ──[superseding ADAPT reached approved via QG-3]──▶ superseded (terminal, immutable)
```

No separate QG is introduced. Supersession goes through the same **QG-3** (§10.8.3) as an ordinary ADAPT — no additional control point is created.

Precondition of the `ADAPT-MMM → superseded` transition:

- A superseding `ADAPT-NNN` has been created with the field `supersedes: ADAPT-MMM` and a non-empty `supersession-rationale` (chapter 7 §7.6.4).
- The superseding ADAPT has passed QG-3 and reached `approved`. If the superseded decision had a contractual outcome (it was a clause of a signed ACTZ) — the cancellation MUST be formalized by a **new ACTZ with a dual signature**, which stamps `superseded-by` on the former one; the Architect signature alone is permitted only for a correction that touches no signed decision.
- All derived `BR` / `SR` / `SPEC` with `source.adapt: ADAPT-MMM` are re-pointed to the superseding ADAPT or re-derived (no dangling references).

Postcondition:

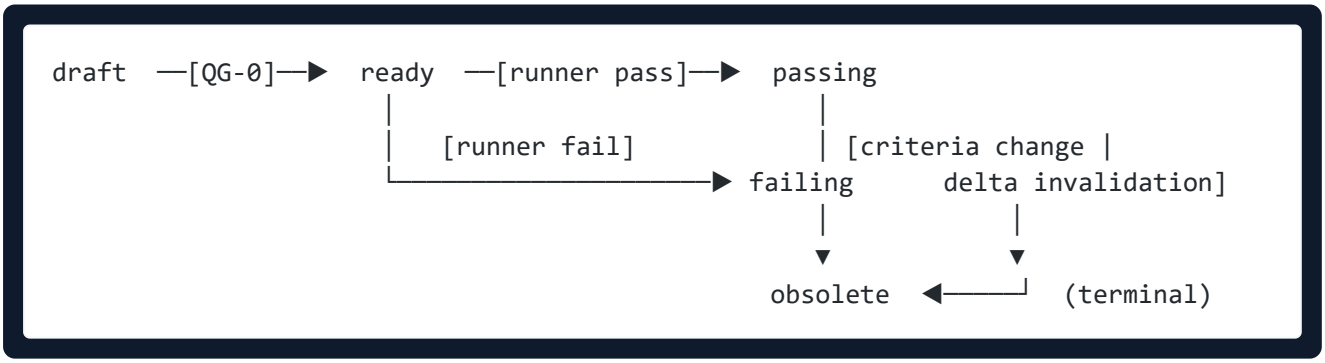
- `ADAPT-MMM` transitions into the terminal `superseded` — distinct from `obsolete` and from `frozen`; immutable and **retained** for audit (V1), not deleted.
- The field `superseded-by: ADAPT-NNN` on `ADAPT-MMM` is recorded automatically.
- A record in the transition log with the signatures of the superseding ADAPT (V6).

Trigger: approval of the superseding ADAPT via QG-3.

Hook obligation: after the transition, a dangling `source.adapt` reference to an ADAPT in status `superseded` is **fatal**; enforcement — the `adapt-supersession` validation (§10.11.1), the `check-adapt-supersession.js` gate.

10.9 TC state machine

10.9.1 States and transitions



Status	Semantics	Gate / trigger
`draft`	TC created; implementation in progress	—
`ready`	dry-run runner passed; pos/neg pairing confirmed	QG-0 (§10.3.1) with the TC preconditions from §10.9.2
`passing`	`last-run.result = pass` on the current `requirement-version`	runner pass; bot-managed
`failing`	`last-run.result = fail`	runner fail; bot-managed

`obsolete`	The covered behavior no longer exists	Deprecation (§10.9.4)

TC frontmatter and pos/neg pairing are defined in chapter 9.

10.9.2 Preconditions for TC

QG-0 for TC (`draft → ready`) — in addition to the common part:

- `automation.status: automated` (with a valid `automation.location`) OR `automation.status: manual-pending` (with `manual-pending-until` ≤ +1 sprint and a filled-in `manual-pending-reason`).
- Pos/neg pairing over the covered assertions confirmed (§9.7).
- The dry-run runner passed (structural validity only; not to be confused with a production run).
- All mandatory body sections of the TC (§9.4) are filled in.
- The citation in `verifies[]` — the artifact exists in the substrate in a state no lower than `approved`.

Postcondition:

- `status: ready`.
- The runner is permitted to do a production run.

10.9.3 Runner-managed transitions (not Quality Gates)

`ready → passing`, `ready → failing`, `passing → failing`, `failing → passing` — transitions that happen **only** upon a runner run (chapter 9 §9.12 `last-run` bot-managed). These transitions are **not** Quality Gates in the sense of §10.2.1: they are normative consequences of run results, not the passing of a gate with preconditions and postconditions. In particular, the check that `last-run.requirement-version` matches the current version of the verified artifact (see §9.10) is a runner-managed consistency check, not a separate gate.

Postcondition of each runner transition:

- `last-run` updated: `result`, `timestamp`, `requirement-version`, `evidence-refs`.
- The substrate MUST prohibit manual modification of `last-run` (runner-actor only).

10.9.4 TC deprecation

A TC transitions into `obsolete` if:

1. An artifact in `verifies[]` transitions into `deprecated` / `obsolete`.
2. A delta-TZ invalidates the test behavior (chapter 9 §9.16).

Postcondition: `status: obsolete`. The TC is not deleted (V1 immutable history).

10.9.5 Change-of-criteria — a separate normative path

Changing `## Pass criterion` or `## Fail criterion` in a TC is **not an ordinary transition**; it is a special path that requires a separate approval workflow (chapter 9 §9.13). Enforcement

details — §10.11.3.

10.10 Closed-list policy

10.10.1 Normative rule

The closed list of RENAR Quality Gates is the mandatory {QG-0, QG-1, QG-2} and the optional {QG-3, QG-4}. Changing the list is possible **only** through the formal RENAR Standard change procedure (chapter 13).

The list closes the **Quality Gates** — the checks on an artifact's transition between states. The mandatory release acceptance gate (AT) (§10.4.3) is not part of it and does not extend it: it normalizes producing the product for trials, not an artifact transition. For the same reason the substrate's transition events (`freeze`, `deprecation`, `runner-pass` / `runner-fail`, `change-of-criteria`) recorded in the log (§10.13.1) stay outside the list.

This policy is a specialization of §1.7 Closed-list policy for Quality Gates; the general rule for all RENAR closed lists and the master index — §1.7.5.

10.10.2 What is prohibited

Action	Prohibited?	Why
Locally creating a new gate type `QG-N` at the project level	Prohibited	Violates the closed list; makes conformance non-portable
Locally overriding the preconditions of a canonical gate	Prohibited	Makes conformance incomparable across implementations
Additionally tightening the preconditions of a local gate	Permitted	The conformance manifest MAY declare stricter thresholds (for example, `qg-2.required-negative-tc: true`)
Locally weakening the preconditions of a canonical gate	Prohibited	Violates the standard's contract
Declaring QG-3 / QG-4 as `absent` in the conformance manifest	Permitted	Optional gates — §10.4
Declaring QG-0 / QG-1 / QG-2 as `absent`	Prohibited	Violates conformance §10.4.4

10.10.3 Extending the list

Adding a new gate type is possible through:

1. A standard change request with a rationale — a research draft with a typology and a comparison with the canonical gates.
2. Public review (the period and forum are fixed by the standard policy, chapter 13).
3. Inclusion in the next minor version of the standard (`v1.X` or `v2.0`).

Project-local extensions remain outside conformance — they are permitted as internal practices but do **not** affect the conformance manifest.

10.11 Substrate-independent enforcement

10.11.1 Normative requirements

A substrate implementing RENAR MUST ensure an automatic check of gate preconditions at the following points:

Control point	What MUST be checked	Relies on capabilities
Promote-transition (any transition to a higher status)	The preconditions of the corresponding gate (§10.3, §10.4, §10.5–§10.9)	V3 (diff & review) to block the transition until approve; V4 (branching) to separate WIP from the approved truth
Approve-transition (any approval action)	Authorship recorded (actor) and timestamp	V6 (author + timestamp)
Reference-validation (any creation/change of an artifact with a reference to another)	The referenced artifact exists and is in the required state	V1 (immutable history) for a stable identifier; V5 (version pin) for cross-substrate references
Change-of-criteria for TC (§10.11.3)	A separate approval process is applied	V3 + V6
Runner-transitions for TC (`ready → passing` / `failing`)	Only the runner-actor may write `last-run`	V6 (authorship); the substrate's native ACL or role-based restrictions
Lifecycle invalidation (artifact `verified` , version incremented)	The artifact is automatically reverted to `approved`	V5 (version pin) for detection
`implements` **-edge validation** (subsystem BR referencing a system BR, §6.5.2, §6.8.2)	(1) the target BR exists by `id + scope.system` ; (2) the target in `approved` + at the approval of this BR (a `deprecated` target — warning, not fatal; cascade-warning over `implemented-by[]`); (3) the `implements` chain forms no cycles; (4) `implements[]` is absent when `level: system`	V1 (stable identifier for target lookup); V3 (block approve until validation passes); V5 (cross-substrate ID resolution when the target is in another substrate)
`actz-integrity` **validation** (§7.13)	(1) An ACTZ in status `signed` carries both signatures (client + supplier), and the signatories are different persons. (2) Every finding in status `resolved` carries a `decided-in` pointing to an ACTZ in status `signed` ; a reference to a	V3 (block approve); V6 (dual signature); V1 (a

	`draft` / `sent` / non-existent ACTZ is fatal. (3) Every signed ACTZ is reflected in at least one ADAPT: a signed decision with no reflection in the interpretation is fatal (an obligation absent from the requirements). (4) `superseded` requires `superseded-by`.	signed ACTZ is immutable)
`at-isolation` validation (§9.19)	(1) The `generator` of each AT attests isolation: the model differs from the model of the main agent, and there was no access to the internal contour — **fatal** on violation (otherwise the AT confirms its own interpretation instead of the contract). (2) `verifies[]` references only `TZ §N` / `ACTZ-NNN §M`; a reference to a BR / SR / SPEC / TC / ADAPT is fatal. (3) The `tz-version` of each AT matches the current revision of the effective TZ — otherwise the trials are blocked (§10.4.3).	V5 (pin to the revision of the effective TZ); V6 (generator provenance)
`red-history` validation (§9.18.2)	(1) A TC counted as evidence at QG-2 has a recorded "red → green" transition, or an inherited red history (a task that does not change behavior). (2) A TC of the `implementation-originated` class has no red history by construction; `mutation-check.mutants-killed ≥ 1` is mandatory for it (§6.13.3). (3) The author of the TC differs from the author of the implementation (P8, §9.18.1).	V1 (the run history is immutable); V6 (authorship provenance)
`adapt-applicability` validation (§7.4.1, §7.4.6)	(1) For each TZ, the adversarial-review verdict is issued as an AR in status `issued` (V6 author + timestamp). (2) If `verdict: findings-present` — `produces-adapt[]` is non-empty, each ADAPT it names exists in `approved` + with the Architect's signature, and every finding of it carries a `decided-in` pointing to a signed ACTZ; the BR/SR/SPEC derivatives have `source.adapt`. (3) If `verdict: no-findings` — no ADAPT, and the BR/SR/SPEC have `source.tz-section` + `source.adversarial-review-ref` pointing at that AR. (4) AR integrity: the `reviewer` model differs from the primary agent's model (§7.10.2); a reference to an AR in status `draft` or `superseded` is fatal; a dangling reference is fatal. (5) No mixing: an artifact with `source.adapt` omitted and without `source.adversarial-review-ref` — fatal.	V3 (block approve until validation passes); V6 (verdict + signature attribution); V1 (AR immutable once `issued`)
`adapt-supersession` validation (§7.6.4, §10.8.5)	(1) `supersedes: ADAPT-MMM` references an existing ADAPT; the back-reference `superseded-by` is symmetric. (2) `supersession-rationale` is non-empty and references a concrete contradicting `BR` / `SR` / `SPEC`. (3) When the superseded decision has a contractual outcome (it was a clause of a signed ACTZ) — the cancellation is formalized by a new ACTZ with the dual signature, which stamps `superseded-by` on the former one. (4) A dangling `source.adapt` reference to an ADAPT in status `superseded` — **fatal**.	V1 (stable identifier + immutable superseded history); V3 (block approve until validation passes); V6 (signature attribution)

10.11.2 A substrate without V3 / V4 / V6 is non-conformant

A substrate that does not provide V3 (diff & review) cannot implement gates: there is no way to separate a "proposed change" from the "approved truth" (chapter 3 §3.3.3). The same holds for V4 (branching, §3.3.4) and V6 (author + timestamp, §3.3.6) — without them the approval mechanics are impossible. A substrate that does not satisfy V3 / V4 / V6 **does not implement RENAR** regardless of other properties.

10.11.3 Change-of-criteria for TC — special enforcement

Changing the Pass / Fail criterion of a TC is a high-risk operation (protection against test-fitting, chapter 9 §9.13). The substrate MUST:

1. **Detect**: any change to the `## Pass criterion` / `## Fail criterion` sections in a TC artifact.
2. **Forcibly isolate**: a change-of-criteria MUST be a separate change-set (V4 branching / change-set, §3.3.4) recorded as an atomic change unit (V2, §3.3.2), marked with a flag distinguishing it from ordinary edits (the substrate's native mechanism — a special case of V3 diff & review; the form of the flag is substrate-specific, deferred to guide/).
3. **Prohibit combining**: the same person MUST NOT approve both a change-of-criteria and a code fix that is tested by this same TC. The substrate MUST check this rule at the approve-transition.
4. **Register**: a change-of-criteria is recorded in the audit trail (§10.13) with explicit event typing.

10.11.4 Forms of substrate-native implementation

The concrete substrate-native mechanisms (how exactly a hook is implemented in a given substrate) are deferred to `guide/` and the conformance manifest. The standard does not normalize the **form** of a hook (this is a substrate-specific decision). The standard normalizes **what a hook MUST check and at which point**.

The `guide/` section MUST contain, for each supported substrate:

- A mapping of the enforcement points of §10.11.1 onto the substrate-native mechanisms.
- Example implementations.
- The known limitations of the substrate regarding the automation of each check.

10.12 Prohibited transitions

A closed list of transitions that violate the lifecycle. The substrate MUST block them.

From	To	Artifact	Why prohibited
`draft`	`verified`	BR / SR / SPEC	Skips QG-0; no approval evidence
`draft`	`accepted`	BR	Same; and skips QG-2
`draft`	`done`	TR	Skips QG-0
`draft`	`passing`	TC	Skips QG-0 (no dry-run runner)
`obsolete`	*	Any	Terminal status; "resurrection" is prohibited — a new artifact with `supersedes` is needed
`deprecated`	*	Any	Same

`frozen`	*	ADAPT	Same; changes only through a delta-ADAPT or errata (§10.8.4)
`verified`	`draft`	BR / SR / SPEC	Degradation across several steps — potential loss of trace; if rework is needed — `verified → approved` via delta or reverse evolution §10.5.4
`accepted`	`verified`	BR	Degradation after acceptance — impermissible without a delta-TZ
`accepted`	`approved`	BR	Same
`passing`	`draft`	TC	Degradation loses run history; use `passing → failing → obsolete` or the change-of-criteria path
`ready`	`draft`	TC	Degradation loses dry-run evidence (§10.9.2); weakening a TC via `[test-spec-change]` (chapter 9 §9.13) — is not a path back to `draft`
`failing`	`draft`	TC	Degradation loses runner history; re-diagnosis is via a new run (`failing → passing` runner-managed) or `obsolete`

10.12.1 Substrate reaction

On an attempt at a prohibited transition the substrate MUST:

1. Block the transition (V3 diff & review).
2. Return to the calling participant an error code indicating the concrete violated rule (by the row identifier of this table).
3. Not create a record in the transition log (§10.13).

10.13 Logging of gate-pass events

10.13.1 Normative requirement

Every successful gate pass (of any type: QG-0, QG-1, QG-2, QG-3, QG-4, runner-transition, deprecation, freeze-transition) MUST be recorded in the substrate as an immutable event with the following fields:

Field	Semantics	Obligation
`timestamp`	UTC ISO-8601 moment of the successful pass	Mandatory
`artifact-id`	Artifact identifier (immutable, V1)	Mandatory
`artifact-type`	`BR` / `SR` / `TR` / `SPEC-<type>` / `ADAPT` / `ACTZ` / `AR` / `TC` / `AT`	Mandatory
`artifact-version`	Artifact version at the moment of the transition (V5)	Mandatory
`from-status`	Source state	Mandatory
`to-status`	Target state	Mandatory

`gate-id`	`QG-0` / `QG-1` / `QG-2` / `QG-3` / `QG-4` / `at-release` (§10.4.3) / `deprecation` / `freeze` / `runner-pass` / `runner-fail` / `change-of-criteria`	Mandatory
`actor`	Initiator identifier (V6); for the ACTZ dual signature — a list of participants	Mandatory
`evidence-refs`	References to evidence: runner run IDs, adversarial-review artifact IDs, signature IDs	Mandatory for QG-2 / QG-3 / QG-4
`notes`	Free text	Optional

10.13.2 Substrate-independent format

The event-storage format is substrate-specific (a separate log stream / append-only collection / other forms). The standard normalizes only the mandatory fields of §10.13.1, not their serialization.

The conformance manifest MUST specify the event-storage mechanism and the export format (for audit, chapter 13).

10.13.3 Retention

Events are **not deleted** throughout the entire artifact lifecycle and after its transition into `deprecated` / `obsolete` / `frozen`. This is required by V1 (immutable history) and the normative compliance clauses (chapter 13).

10.14 Relationship to other chapters

Chapter	Relationship
02	SENAR QG-0..QG-2 — the conceptual basis; RENAR extends it (§10.2.3)
06	frontmatter and body of BR / SR / TR (§6.5–§6.7); the state machines are detailed here (§10.5–§10.6)
07	ADAPT frontmatter (§7.8); backward sub-states (§7.4.5); the Architect's signature (§7.5); ACTZ and the dual signature (§7.13); delta-ADAPT (§7.6) — the state machine is here (§10.8)
08	SPEC frontmatter (§8.4); type-specific QG (§8.8); the state machine is here (§10.7)
09	TC frontmatter (§9.3); pos/neg pairing (§9.7); change-of-criteria (§9.13); the state machine is here (§10.9)
03	V1–V6 — the foundation of enforcement (§10.11); without V3 / V4 / V6 — no gate implementation
11	The maturity levels determine the scope of applicable gates (for example, RENAR-1 — QG-0 / QG-1 / QG-2 mandatory; at higher levels the QG-2 preconditions are strengthened: pos/neg pairing, spec-specific TC)
13	The conformance manifest declares gate support (§10.4.4); holds the event retention policy (§10.13.3)

11. Maturity Model

> ***Part of the RENAR Standard v1.0*** · *← Table of contents*

11.1 Maturity as a ladder, not a label

Chapter 3 laid the foundation — a substrate capable of storing the history and provenance of requirements. But having a foundation says nothing about how maturely a team uses it. One project simply keeps its requirements in a substrate; another validates them against a schema, runs paired tests, and makes a second model challenge the first. These are different rungs of one ladder — and this chapter numbers them.

RENAR defines **five maturity levels** — `RENAR-1` .. `RENAR-5`, a closed list. A level is a **measurable characteristic** of how formalized requirements management is in a project. It is important not to confuse it with a **conformance claim** (conformance): the level describes *where* a project stands, while conformance is the procedure by which it *proves* this. The mechanics of the claim are in chapter 13.

Each next rung adds obligations on top of the previous one and closes its class of divergences; the chapter is best read in order: §11.4–§11.8 cover the rungs one at a time, §11.10–§11.11 answer "where to start" and "how to grow."

The chapter's boundary is strict. It governs **only** the level criteria and the transitions between them. The conformance claim procedures are chapter 13; the metrics for measuring a level are chapter 12; the substrate capabilities themselves are chapter 3.

11.2 RENAR-M as a domain-specific dimension in the SENAR maturity model

11.2.1 The SENAR maturity model (general maturity)

SENAR defines a one-dimensional model of five levels of the team's and methodology's general maturity: `Ad-hoc → Supervised → Measurable → Managed → Optimizing`. This model assesses the **process maturity of the team as a whole**, independently of any specific process area.

11.2.2 RENAR-M as a separate dimension

RENAR-M is a **separate dimension** within the general SENAR maturity, specializing it for the "requirements engineering" process area. A project is characterized by a **pair**:

```
project ──▶ (SENAR-N, RENAR-M)

where N ∈ {1, 2, 3, 4, 5} — general SENAR maturity
      M ∈ {1, 2, 3, 4, 5} — RENAR level of requirements management
```

The pairs `(SENAR-N, RENAR-M)` are normatively independent: a project at SENAR-4 (Managed) MAY be at RENAR-2 (Documented) — the team is mature in SENAR practices overall, yet **requirements management specifically** is weakly formalized. This is a permitted and observable scenario.

11.2.3 Alignment with the SENAR Reference

The SENAR Reference allows domain-specific maturity dimensions (authentication, security, observability, and other process areas). RENAR-M is one such dimension; it does not contradict SENAR and does not duplicate it.

In the industry it is typical for different process areas within one organization to have differing maturity. RENAR-M is the normative maturity dimension specifically for the "requirements engineering" area, with criteria drawn only from this chapter and adjacent sections of the standard.

11.2.4 Conformance as a pair

The conformance manifest (§13.4) records **only** the RENAR-M (the `level` field). SENAR-N remains within the scope of SENAR conformance and is not duplicated in the RENAR manifest.

11.3 The closed list of levels RENAR-1..RENAR-5

The list of five levels is closed. Changing the list is possible only through the formal change procedure of the standard (§13.9.3).

Level	Short name	Semantics (one line)
`RENAR-1`	Ad-hoc	A requirements substrate exists; artifacts are kept without a formal schema or lifecycle
`RENAR-2`	Documented	Artifacts have basic frontmatter and are stored structurally; the TZ is fixed
`RENAR-3`	Tracked	Full frontmatter schema; lifecycle statuses are used; delta-TZ workflow via ADAPT
`RENAR-4`	Verified	100% of `approved` have `verified-by`; pos/neg pairing; QG-2 is enforced; AI provenance
`RENAR-5`	Optimized	Adversarial review as a gate; multiple models for `priority: must`; knowledge graph; metrics

Intermediate levels (`RENAR-3.5`) and project-local levels are prohibited (§13.9.2). Local tightening of criteria within a declared level is permitted via `declared-stricter` (§13.4.2).

Each level includes the normative criteria of all lower ones. `RENAR-M` implies satisfaction of the requirements of `RENAR-1` .. `RENAR-(M-1)` .

11.4 RENAR-1: Ad-hoc

The first rung answers the question: **where do the requirements live?** The answer is "in the substrate, not in someone's head, an issue tracker, or a chat thread." There is no formal schema or lifecycle yet, but provenance is already recoverable.

11.4.1 Normative definition

RENAR-1 is the lowest conformant level. A project MUST satisfy: the substrate exists physically (V1–V6 §13.3.2 — an absolute mandatory clause regardless of level); requirements are kept as artifacts inside the substrate (not in issue trackers or chat threads); each TZ has passed the adversarial review, its outcome is issued as an AR, and an ADAPT is created on a "findings present" verdict (§13.3.3); substrate-independent language is applied (§2.5.4).

> ***RENAR-1 ≠ zero infrastructure.*** *"Ad-hoc" is about the absence of a formal schema / lifecycle, not of a substrate: V1–V6 are mandatory at any level. A flat file server, Notion, Google Docs, or Confluence without immutable history / atomic change / version pin* ***do not qualify*** *even for RENAR-1. The minimum is a distributed VCS or a document-oriented store with V1–V6 (§3, guide/03–04).*

11.4.2 What is NOT required at RENAR-1

Standardized frontmatter (a minimum of `id` + `title` is acceptable); lifecycle statuses; TC as separate artifacts (keeping them in code is acceptable); COVERAGE; substrate-native lifecycle hooks.

11.4.3 Observable signals

BR/SR/ADAPT artifact files in the substrate (not in a tracker/chat); on a "where does this requirement come from" query, the answer is given through the substrate; a manifest (§13.4) with `level: RENAR-1`.

11.4.4 QG application (enforcement)

QG-0/QG-1/QG-2 are normatively required (§13.3.6), but enforcement through hooks is **not mandatory** — a human checks manually as needed.

11.5 RENAR-2: Documented

RENAR-2 adds **structure** on top of existence: basic frontmatter, the TZ as an immutable artifact, a predictable location for artifacts. A requirement can be found, quoted, and referenced. There is no machine checking yet; discipline is held by people.

11.5.1 Normative definition

On top of RENAR-1: every BR/SR has frontmatter with the mandatory fields (§6.5.2, §6.6.2) — at minimum, without strict schema validation; the TZ is an immutable artifact (§7.4.2); structural storage (logical folders / native collections); a delta-TZ is an explicit artifact, not a verbal one.

11.5.2 What is NOT required at RENAR-2

CI validation of frontmatter against a schema; the full lifecycle (statuses are at the team's discretion); the `tc-type` TC extension; pos/neg pairing of TC.

11.5.3 Observable signals

All BR/SR have valid (at-minimum) frontmatter; the TZ is one native artifact; the delta-TZ is an explicit change-set (§7.6); a manifest with `level: RENAR-2`.

11.5.4 QG application (enforcement)

QG-0 (§10.3.1) is a procedural gate (an explicit approval with a V6 author + timestamp), without automatic blocking hooks.

11.6 RENAR-3: Tracked

At RENAR-3 the **machine** kicks in. Frontmatter is validated against a schema, lifecycle statuses work, and the delta-TZ goes through ADAPT. The substrate blocks a reference to a non-approved ADAPT and does not let an implementation reference a requirement without a version binding.

11.6.1 Normative definition

On top of RENAR-2: frontmatter is validated against the schema (reference/02-schemas.md) by a native mechanism (§10.11.1); lifecycle statuses are actually used (chapter 10); TC exist for all BR/SR/SPEC with `priority: must`; COVERAGE is auto-generated (§9.15); the delta-TZ is a change-set with impact analysis (§9.16); the reference-validation hook (§10.11.1) blocks the creation of an artifact referencing an ADAPT below `approved`; the implementation substrate binds `verifies[].requirement-version` (§9.3, V5).

11.6.2 Observable signals

The frontmatter-validation hook returns a structured response on a schema violation; every artifact $\in$ { `draft`, `approved`, `verified`, `deprecated`, `obsolete` } (§10.5); COVERAGE is updated within a reasonable time; on a delta-TZ the architect automatically sees the affected SR/SPEC/TC; a manifest with `level: RENAR-3`.

11.6.3 QG application (enforcement)

QG-0 + QG-1 are enforced natively; QG-2 partially (the `verifies[]` check is mandatory, pos/neg pairing (§9.7) is not required to be automatic).

11.7 RENAR-4: Verified

RENAR-4 is the threshold of **trust in tests**. Every normative statement is covered by a pos/neg TC pair, QG-2 blocks promotion to `verified` without green tests on the current version, and a once-per-iteration spot-check catches tests faked to match the code.

11.7.1 Normative definition

On top of RENAR-3: 100% of artifacts in `approved` have `verified-by` referencing ≥ 1 TC (§9.3); pos/neg pairing (§9.7) is done for **every** normative statement (single-TC coverage is permitted only for invariants with negative semantics); QG-2 (§10.3.3) is **enforced** — promotion to `verified` is blocked unless all TC are `passing` on the current `requirement-version`; all TC are automated (`automation.status: automated`) or marked `manual-pending` with a deadline (§9.5); for `tc-type: ux` — VLM judge isolation (§9.13.4); for `tc-type: eval` the judge model differs from the implementation model (Decision #8); AI provenance in frontmatter (§4.10.1): at minimum `generated-by` and `generated-at` are mandatory; source citation in the artifact body (`[TZ-XXX §Y line Z]` or a `derived` marker); a continuous-reconciliation hook (§2.4.2) on a schedule (no less than once a week); a spot-check of 5 random passing TC once per iteration (§9.14) in the audit-trail.

11.7.2 Observable signals

COVERAGE contains `verified-by-percent: 100%` for `approved`; an attempt to promote a BR/SR/SPEC to `verified` without all passing TC returns a blocking error; a sample of 5 random artifacts shows source citation on all normative statements; a manifest with `level: RENAR-4`.

11.7.3 QG application (enforcement)

QG-0/QG-1/QG-2 are enforced natively. QG-3 (Architecture, §10.4.1) is declared if the project uses ADAPT with the Architect's signature and ACTZ with the dual signature (the default for regulated industries).

11.8 RENAR-5: Optimized

RENAR-5 closes the loop of **AI reliability**. A second model is set to challenge the first, critical requirements are generated by several models with mandatory analysis of divergences, and the hallucination rate is held below 1%. The standard becomes a system that checks itself.

11.8.1 Normative definition

On top of RENAR-4: **adversarial review** is a mandatory gate for `draft → approved` (the artifact passes review by a second AI model, recorded in the audit-trail); **multi-model agreement** for `priority: must` (the artifact is generated by ≥ 2 models; divergences are flagged `[multi-model-disagreement]` and MUST be analyzed); a **cost/latency budget** per artifact (`cost-budget` + `latency-budget`; exceeding it triggers automatic decomposition); a **knowledge graph** (reference/05) as the primary search for AI agents; **continuous evaluation** for all SPEC-AI (§8.5.7); the **Hallucination Rate** (chapter 12) is measured and < 1%; the **Multi-model Disagreement Rate** (chapter 12) is measured and its trends are tracked; feeding template improvements back into the `requirements-library` is standard practice.

11.8.2 Observable signals

Every promoted artifact has an adversarial-review record in the audit-trail; for `priority: must` BR — a `[multi-model-agreement]` or `[multi-model-disagreement]` marker; a

knowledge-graph dashboard is available; a Hallucination Rate dashboard shows < 1% over a rolling window; a manifest with `level: RENAR-5`.

11.8.3 QG application (enforcement)

All mandatory gates (QG-0/1/2) are enforced natively. Adversarial review is enforced as part of QG-0 — the substrate blocks `draft → approved` without confirmation. QG-3 is declared. QG-4 is declared if the project applies post-release outcomes (§10.4.2).

11.9 Comparative feature table

Everything spread across the five sections above, in one matrix. Cells: ❌ — not required; `partial` — required, but without substrate enforcement (the check is procedural); ✓ — normatively mandatory.

Feature	RENAR-1	RENAR-2	RENAR-3	RENAR-4	RENAR-5
Substrate with V1–V6	✓	✓	✓	✓	✓
Adversarial review of the TZ with an issued AR; ADAPT per the verdict	✓	✓	✓	✓	✓
Frontmatter standardized	❌	partial	✓	✓	✓
Lifecycle statuses used	❌	partial	✓	✓	✓
Frontmatter schema validation (substrate hook)	❌	❌	✓	✓	✓
TC as a full-fledged artifact	❌	❌	partial	✓	✓
Pos/neg pairing for every statement	❌	❌	❌	✓	✓
COVERAGE auto-generated	❌	❌	✓	✓	✓
Reference-validation hook	❌	❌	✓	✓	✓
`verifies[].version` binding (V5)	❌	❌	✓	✓	✓
QG-0 enforced natively by the substrate	❌	partial	✓	✓	✓
QG-2 enforced natively by the substrate	❌	❌	partial	✓	✓
AI provenance in frontmatter	❌	❌	❌	✓	✓
Source citation in artifact bodies	❌	❌	❌	✓	✓
Adversarial review as a gate	❌	❌	❌	❌	✓
Multi-model agreement for `priority: must`	❌	❌	❌	❌	✓
Cost and latency budget per artifact	❌	❌	❌	❌	✓

Knowledge graph as primary search	❌	❌	❌	❌	✓
Continuous reconciliation	❌	❌	❌	✓ (basic)	✓ (full)
Continuous evaluation (SPEC-AI)	❌	❌	❌	❌	✓
Hallucination Rate < 1%	n/a	n/a	n/a	n/a	measured and controlled
Multi-model Disagreement Rate trend	n/a	n/a	n/a	n/a	tracked

11.10 Minimum entry level (entry-level)

`RENAR-1` is the normative **lower bound** of the conformance manifest. A project that does not satisfy the mandatory clauses (§13.3) — in particular, without a substrate with V1–V6, or without a recorded adversarial-review verdict (AR) for each TZ — has **no** RENAR-M level at all; a conformance manifest is not issued for such a project.

Project type	Recommended target level
Short experiment (spike), < 1 sprint, no contract	RENAR-2 — sufficient for documentation
Internal automation, 1–3 months	RENAR-3 — tracked lifecycle
Client product under contract	RENAR-4 — verified, with pos/neg pairing
AI-critical component (depends on eval)	RENAR-5 — mandatory
Regulated industries (medicine, fintech, the public sector)	RENAR-4 minimum, RENAR-5 recommended

The standard allows **different levels in different projects** of one organization — there is no requirement to unify the level across a portfolio.

Negative scenario: a substrate that lacks even one capability from V1–V6 (§3.2) cannot normatively implement any RENAR-M — not even `RENAR-1`. This is a structural constraint, not an operational one; the manifest of such a project is invalid (§13.3.2).

11.11 The path RENAR-1 → RENAR-5

The normative sequence of steps between adjacent levels. The times are an expected order of magnitude (for a small project; this is not a normative guarantee).

11.11.1 RENAR-1 → RENAR-2

1. Create structural storage of artifacts in the substrate (logical folders or substrate-native collections).
2. Migrate existing requirements from the issue tracker, chat, or documents into substrate-native artifacts with minimal frontmatter (`id` , `title`).
3. Fix the TZ as an immutable artifact (§7.4.2).
4. Agree within the team: new requirements only through the substrate, not through the issue tracker.

Expected time: 1–2 weeks for a small project; 1–2 months for a project with a large volume of accumulated requirements.

11.11.2 RENAR-2 → RENAR-3

1. Bring the frontmatter of all artifacts into line with the schema (a substrate-native normalizer).
2. Enable the substrate hook for schema validation (block the change-set on a violation).
3. Introduce the lifecycle: go through all artifacts and assign statuses.
4. Enable the reference-validation hook (§10.11.1).
5. Generate the initial auto-generated COVERAGE artifact (§9.15).
6. Create TC for artifacts with `priority: must`.
7. Introduce `verifies[].version` binding from the implementation substrate.

Expected time: 2–4 weeks, including team training.

11.11.3 RENAR-3 → RENAR-4

1. An AI agent goes through all artifacts and generates pos/neg TC pairs for every normative statement.
2. Implementation of TC in code / SPEC-runners — in parallel with product work; spread over 1–2 quarters.
3. Enable the QG-2 substrate hook (block promotion to `verified` without all passing TC).
4. Introduce the spot-check process (§9.14).
5. Enable the continuous-reconciliation hook on a weekly schedule.
6. Introduce AI provenance in frontmatter (the substrate hook blocks when it is missing for AI-generated artifacts).
7. Introduce source citation in artifact bodies (the substrate hook blocks when a citation is missing for normative statements).

Expected time: 1–2 quarters.

11.11.4 RENAR-4 → RENAR-5

1. Connect adversarial review by a second model (different from the primary one; the judge-isolation principle §9.13.4); enable it as QG-0 enforcement.
2. Enable generation by several models for artifacts with `priority: must`.
3. Deploy the knowledge graph (reference/05).
4. Set up the targeted Hallucination Rate and Multi-model Disagreement Rate metrics (chapter 12).
5. Introduce a cost and latency budget with automatic decomposition on overrun.
6. Establish the practice of feeding template improvements back into the `requirements-library`.
7. Continuous evaluation for all `SPEC-AI` artifacts (§8.5.7).

Expected time: 1–2 quarters after RENAR-4.

11.11.5 Reverse transitions (level downgrade)

A normative downgrade (§13.8.2) — issuing a new manifest version with a level lower than the current one — is permitted given a formal justification (for example, decommissioning an AI-critical component,

simplifying the substrate). A downgrade MUST be accompanied by an audit-trail record; concealing a downgrade is a violation of a mandatory clause (§13.3.1).

11.12 Relationship to the SENAR ADR (Adversarial Detection Rate)

SENAR §9 defines ADR — Adversarial Detection Rate — as a general metric of a process's ability to detect errors before release to production. RENAR-M defines at which levels a normative adversarial-review infrastructure exists. The RENAR-specific subclass of ADR for the requirements zone is **ACR** (Adversarial Catch Rate, §12.3.6).

RENAR-M	Normative adversarial-review infrastructure	ADR / ACR measurability
RENAR-1, RENAR-2	Absent (§11.4, §11.5)	n/a
RENAR-3	Normatively absent; the team MAY introduce adversarial review as a local tightening (`declared-stricter` , §13.4.2)	optional
RENAR-4	Pos/neg TC pairing (§9.7) — a structural proxy for ADR; adversarial review of artifacts is not normatively required	optional (pos/neg coverage is measurable as a proxy)
RENAR-5	Adversarial review as a mandatory gate (§11.8.1) + multi-model agreement for `priority: must`	normatively measurable (ACR, §12.3.6)

RENAR-M maturity is linked to the SENAR ADR metric **continuously**, without duplication: SENAR ADR sets the concept of the metric; the RENAR-M level defines which adversarial cycles are normative; ACR (§12.3.6) is the domain-specific metric for the requirements zone.

11.13 Relationship to other chapters

The maturity model is the map onto which the requirements of all the other chapters are placed: each chapter "turns on" at its own level. The table below shows exactly where.

Chapter	Relationship
02 Positioning in the methodology typology	§2.3 Source-of-Truth inversion + §2.5 substrate-independent versioning — mandatory clauses regardless of level; §2.4 the four distinctions — observed at all levels, starting with RENAR-2
06 Requirements hierarchy	artifact frontmatter and mandatory fields are checked at RENAR-3+; the BR/SR/TR hierarchy — at all levels
07 ADAPT	Reactive ADAPT — a mandatory clause regardless of level (§11.4.1): an adversarial review for each TZ, ADAPT per the verdict; the Architect's signature on an ADAPT (QG-3) — declared at RENAR-4+; the dual signature is on the ACTZ (§7.13)
08 Specifications	The closed list of 11 SPEC types — mandatory; full type-specific coverage at RENAR-4+; continuous evaluation of SPEC-AI at RENAR-5
09 Test cases	TC for `priority: must` at RENAR-3; pos/neg pairing at RENAR-4+; the spot-check process at RENAR-4+

10 Lifecycle and QG	QG-0/QG-1/QG-2 declared required at all levels; substrate enforcement is gradual — partial at RENAR-2, full at RENAR-4+; QG-3 declared at RENAR-4+ if ADAPT is used
03 Substrate versioning	V1–V6 — an absolute mandatory clause for any level; a substrate without V1–V6 has no RENAR-M level (§11.10)
12 Metrics	the Hallucination Rate / Multi-model Disagreement Rate / Defect Escape Rate metrics are measured at RENAR-4+; the precise definitions are in chapter 12
13 Conformance	§13.2 references this chapter for the criteria of each level; §13.5 self-assessment uses the checklists of §§11.4–12.8 (one section per level); §13.9 the closed-list policy applies to the closed list RENAR-1..5 (§11.3)

12. Metrics

> ***Part of the RENAR Standard v1.0*** · *← Table of contents*

12.1 What to measure in requirements

The general SENAR metrics (§9) will show that the team is fast and the tests are green. But they will not notice an AI agent quietly inserting into an SR a clause that was in neither the TZ nor the ADAPT — until it surfaces at acceptance as a dispute with the client. "The process as a whole" is healthy, yet the requirements work is not. So RENAR adds **eleven metrics that look specifically at requirements**: how often an AI agent invents a clause without a source (Hallucination Rate), whether paired tests catch real defects, how quickly a delta-TZ reaches the code. This is an overlay on SENAR §9, not a replacement.

The list of metrics is closed; for each one, a formula, a target per maturity level (chapter 11), and a data source are specified. Metrics are collected natively for the substrate through V1–V6 (chapter 3); exactly how to render dashboards is an implementation question deferred to `guide/`. This chapter does not duplicate SENAR §9 but specializes it, and it does not touch ROI or pricing — those are not process indicators but business effects (§12.5).

12.2 Relationship to SENAR §9

SENAR §9 defines ten general-process metrics: Throughput, Lead Time, FPSR (First-Pass Success Rate), DER (Defect Escape Rate), KCR (Knowledge Capture Rate), Cost Predictability, Cost-per-task, MIR (Memory Integrity Rate), Cycle Time, ADR (Adversarial Detection Rate).

RENAR §12 does **not** edit and does **not** replace these metrics. The REQ-specific metrics of §12.3:

- **Refine** a SENAR metric for the requirements phase (for example, RDLT refines SENAR Lead Time for the requirements phase).
- **Add** observations specific to requirements engineering and not covered by SENAR §9 (Hallucination Rate, Multi-model Disagreement Rate).

The full mapping — §12.7.

The closed list of REQ metrics (§12.3) is maintained within RENAR; SENAR §9 is a separate closed list of general metrics. Changing either of the two lists is a formally independent change procedure of the respective standard.

12.3 Closed list of REQ-specific metrics

A closed list of eleven REQ metrics. The list is changed only through the formal change procedure of the standard (§13.9.3); the general closed-list policy and master index — §1.7.5.

12.3.1 RDLT — Requirement Decomposition Lead Time

The time from registering a TZ in the substrate to the state "all BR/SR (the parent chain from this TZ) → `approved`, ready for QG-0 (§10.3.1)". **Formula:** `RDLT = timestamp(last BR/SR →`

`approved) – timestamp(TZ registered)`. Measured in hours/days. **Source:** the audit log of promote-transitions (§10.13). **Relationship to SENAR:** a refinement of SENAR `Lead Time` for the requirements phase. **Targets:** RENAR-3 < 1 week per 50-page TZ; RENAR-4 < 2 days; RENAR-5 < 4 hours.

12.3.2 Requirement-to-Task Latency

The time from the promote-transition SR → `approved` to the creation of the first TR with the reference `implements: SR-N`. **Formula:** `Latency = timestamp(first TR.created) – timestamp(SR → approved)`. Hours. **Source:** the substrate audit log + the cross-substrate references of the implementation substrate. **Relationship to SENAR:** a refinement of SENAR `Cycle Time` for the "requirement → executable task" pair. **Targets:** RENAR-3 < 3 days; RENAR-4 < 1 day; RENAR-5 < 1 hour (auto-create TR after approval).

12.3.3 Hallucination Rate

The percentage of normative assertions in an AI-generated artifact (BR/SR/SPEC) that are not traceable to a source (TZ/ADAPT/another normative artifact). Source citation is checked by a native citation parser (§13.3.1, REQUIRED at RENAR-4). **Formula:** `assertions_without_valid_citation / total_normative_assertions × 100%`. Per artifact; aggregated per project. **Source:** the citation parser (AST or regex over inline references `[TZ-XXX §Y]` / `[ADAPT-NNN §Z]`). **Relationship to SENAR:** a new metric; corresponds to ISO/IEC 5338 traceability for AI-generated artifacts. **Targets:** RENAR-1..3 n/a; RENAR-4 ≤ 5%; RENAR-5 ≤ 1%.

Negative scenario (loss-of-conformance trigger): a Hallucination Rate > 5% on a RENAR-4 project is a normative loss-of-conformance trigger (§13.8.1); remedy it through a release recovery plan or downgrade to RENAR-3.

12.3.4 Multi-model Disagreement Rate

The percentage of artifacts with `priority: must` where two (or more) generating AI models produced normative assertions diverging beyond a threshold (by default, embedding similarity < 85%; the threshold is fixed in the manifest under `declared-stricter`). **Formula:** `BRs_with_high_disagreement / total_must_BRs × 100%`. **Source:** the multi-model runs log; embedding similarity computed offline over "model A vs B" pairs. **Relationship to SENAR:** a new metric. **Targets:** RENAR-1..4 n/a; RENAR-5 tracked, with per-project baseline values in the first quarter. **Interpretation:** a high value is an indicator of weak prompt engineering or a complex problem domain; it warrants attention but is not in itself a negative indicator.

12.3.5 DRA — Dispute Rate at Acceptance

The percentage of BR/SR for which, at the QG-4 stage (§10.4.2), the client declared disagreement with the interpretation or coverage. **Formula:** `disputed_BRs_at_QG4 / total_BRs_in_release × 100%`. **Source:** the QG-4 audit log — a `gate-id: QG-4` event with `result: disputed`. **Relationship to SENAR:** a refinement of `DER (Defect Escape Rate)` for requirements. **Applicability:** only when QG-4 is in the manifest; when QG-4 = `absent` (§13.3.6) — not measured. **Targets:** RENAR-3 ≤ 10%; RENAR-4 ≤ 5%; RENAR-5 ≤ 2%.

12.3.6 ACR — Adversarial Catch Rate

The percentage of artifacts (BR/SR/SPEC) where the AI critic (a different model; REQUIRED at RENAR-5 per §11.8.1) found ≥ 1 high-severity issue before QG-0. **Formula:** `artifacts_with_critic_high_findings / total_reviewed_by_critic × 100%`. **Source:** the critic-runs audit log; severity (`high` / `medium` / `low`) in the critic output. **Relationship to SENAR:** a subclass of `ADR (Adversarial Detection Rate)` for requirements. **Targets:** RENAR-1..3 n/a; RENAR-4 optional (if `declared-stricter` — a baseline level of 20–30%; values < 20% are an indicator of a weak critic or of duplicating the primary); RENAR-5 tracked normatively, a rising ACR warrants attention to prompt quality.

12.3.7 Test-spec Drift Rate

The percentage of TC in status `passing` (§10.9.1) whose `last-run.requirement-version` (§9.12) differs from the current `version` of the verified artifact (`verifies[]`). **Formula:** `stale_passing_TCs / total_passing_TCs × 100%` (stale = `last-run.requirement-version` < current). **Source:** the COVERAGE artifact (§9.15). **Relationship to SENAR:** a new metric. **Targets:** RENAR-1..3 n/a; RENAR-4 ≤ 5%; RENAR-5 ≤ 1% (auto-rerun on delta-ADAPT).

12.3.8 Coverage Velocity

The rate of `approved` → `verified` transition (§10.5) per unit of time (default iteration; the substrate MAY define a different interval in the manifest). **Formula:** `(verified_count(end) − verified_count(start)) / approved_count(start) × 100%`. **Source:** the COVERAGE artifact history (§9.15). **Relationship to SENAR:** a refinement of `Throughput` for requirements. **Targets:** RENAR-3 ≥ 30%/iteration; RENAR-4 ≥ 50%/iteration; RENAR-5 ≥ 70%/iteration.

12.3.9 Cost per Approved Requirement

The AI-generation cost (input tokens + output tokens × tariff) normalized to one artifact in status `approved`, including rejected versions (which remain in the substrate by virtue of V1 immutable history). **Formula:** `total_AI_tokens_cost(period) / count(artifacts_approved_in_period)`. Project currency. **Source:** the `ai-provenance.cost-budget` + `ai-provenance.cost-actual` frontmatter fields (§11.7.1). **Relationship to SENAR:** a refinement of `Cost-per-task` for requirements. **Targets:** RENAR-1..3 n/a; RENAR-4 tracked, with per-project baseline values; RENAR-5 tracked + year-over-year reduction or a justification.

12.3.10 Reconciliation Findings per Week

The number of issues detected by the reconciliation agent (§2.4.2 continuous reconciliation) per week and registered as backward findings in a delta-ADAPT or a direct change-set requirement. **Formula:** `count(reconciliation_findings_registered) / weeks_in_period`. **Source:** the reconciliation-runs audit log + the ADAPT backward findings list (§7.4.5). **Relationship to SENAR:** a new

metric. **Targets:** RENAR-1..3 n/a; RENAR-4 tracked > 0 (zero = the reconciliation hook is not working or V5 is lost); RENAR-5 trending **down** over the long term (a mature process does not generate new drift).

12.3.11 Implementation-originated Rate

The share of requirements of the class `source: implementation-originated` (§6.13) among all requirements added over the period. **Formula:** `IOR = count(SR where source=implementation-originated) / count(all SR added) × 100%`. **Source:** the `source` frontmatter field + the audit log.

Why the metric is mandatory. The `implementation-originated` class is a narrow legalization of internal technical details that arise during implementation. It is useful while it stays rare: its growth means that **the requirements process is leaking** — functionality is born in the code rather than in the requirements, and is legalized after the fact. The norm of §6.13 requires this drift to be **visible**, not smeared imperceptibly across the project.

Targets: RENAR-3 ≤ 15%; RENAR-4 ≤ 5%; RENAR-5 ≤ 2%. Exceeding them is not in itself a breach of conformance, but it MUST be examined at the review (§12.5.3): either the requirements are written insufficiently completely, or the "observable by the client" boundary is being construed too broadly.

12.4 Summary table of targets by level

The closed list of 11 metrics from §12.3 — target values for the applicable levels.

Metric	RENAR-3	RENAR-4	RENAR-5	Data source
RDLT (Decomposition Lead Time)	< 1 week	< 2 days	< 4 hours	promote-transitions audit log
Requirement-to-Task Latency	< 3 days	< 1 day	< 1 hour	audit log + cross-substrate refs
Hallucination Rate	n/a	≤ 5%	≤ 1%	citation parser
Multi-model Disagreement Rate	n/a	n/a	tracked	multi-model runs log
DRA (Dispute Rate at Acceptance)	≤ 10%	≤ 5%	≤ 2%	QG-4 audit log (if QG-4 declared)
ACR (Adversarial Catch Rate)	n/a	optional (if critic declared-stricter)	tracked normatively	critic-runs audit log
Test-spec Drift Rate	n/a	≤ 5%	≤ 1%	COVERAGE artifact
Coverage Velocity	≥ 30%/iteration	≥ 50%/iteration	≥ 70%/iteration	COVERAGE history
Cost per Approved Requirement	n/a	tracked	tracked + optimized	ai-provenance fields
Implementation-originated Rate	≤ 15%	≤ 5%	≤ 2%	frontmatter `source` + audit log

Reconciliation Findings/Week	n/a	tracked (> 0)	trending down	reconciliation audit log

The concrete substrate-native presentation of these metrics in dashboards is substrate-specific and deferred to `guide/`.

> *The target values for RENAR-4 / RENAR-5 are **provisional**: set as normative direction markers, but subject to calibration against field data in version v1.1 (see guide/07 §8.1). These are guiding thresholds, not statistically validated values.*

12.5 Business outcomes

Six normative effects of adopting RENAR, expected at the RENAR-3 level and above. The outcomes are **normative expectations of the standard**, not process indicators; measuring them is substrate-specific and not mandatory (although the §12.3 metrics capture them indirectly).

> *ROI / cost-of-adoption is a **non-normative** topic and is not part of a normative chapter. A lightweight **illustrative** (not guaranteed) model of the cost and benefit of adoption for a decision-maker — in guide/02-transition-guide.*

12.5.1 Outcome 1 — Reduced TZ decomposition time

Measured through §12.3.1 RDLT. Expected reduction from the baseline: on the order of 5–10× at RENAR-4, 20–50× at RENAR-5.

12.5.2 Outcome 2 — Lower dispute frequency at acceptance

Measured through §12.3.5 DRA. Expected reduction: from 15–30% to ≤ 5% at RENAR-4, ≤ 2% at RENAR-5.

12.5.3 Outcome 3 — Audit readiness

The event audit log (§10.13) + AI provenance in frontmatter (§11.7.1) provide a compliance audit without separate preparatory work. Applicable to regulated industries (medicine, fintech, the public sector).

12.5.4 Outcome 4 — Lower cost of working with a delta-TZ

Impact analysis (§9.16) + reverse evolution of verification (§10.5.4) automate the handling of a delta-TZ. Expected reduction in human involvement: from tens of hours to an hour per delta.

12.5.5 Outcome 5 — Lower knowledge loss on team turnover

V6 author + timestamp (§3.3.6) records the author of all artifacts; the Source-of-Truth inversion (§2.3.1) moves knowledge out of people's heads into the substrate. Expected reduction in onboarding: from weeks to days.

12.5.6 Outcome 6 — The standard as a sellable product

At RENAR-4/5 a substrate-native implementation MAY be licensed to partners as an actionable product. A structural consequence of a formal standard, not a process metric.

12.6 Substrate-independent metric collection

12.6.1 Normative requirements

A substrate implementing RENAR at the RENAR-4+ level MUST provide **automatic collection** of the §12.3 metrics through:

Source	Capabilities	Accessible
Event audit log (§10.13)	V1 + V6	gate-passage events with timestamps, artifact-version, actor
COVERAGE artifact (§9.15)	V5 + V1	counts approved/verified/total; pos/neg coverage; stale-rate
AI-provenance frontmatter fields	V6	cost-budget, cost-actual, generated-by, generated-at
Reconciliation audit log	V1 + V6	reconciliation runs + findings list ID
Critic-runs audit log (RENAR-5)	V1 + V6	critic runs + severity classifications

A substrate without access to any of the sources above **cannot** implement RENAR-4/5 (§11.7.1, §11.8.1).

12.6.2 Substrate-native monitoring panels (dashboards)

The format (UI/CLI/report-generation) is substrate-specific; the standard does not regulate visualization. The substrate MUST export metrics in a machine-readable format for external audit (§13.6 third-party assessment). Dashboard templates — `guide/` .

12.6.3 Period aggregation

Per artifact — Hallucination Rate, Cost per Approved Requirement; per period (sprint/week/month) — Coverage Velocity, Reconciliation Findings/Week, ACR; per release — DRA; continuous trending — Multi-model Disagreement Rate, Test-spec Drift Rate. Period boundaries are fixed in the manifest (§13.4.2) under `declared-stricter` or are taken by default.

12.7 Mapping to SENAR metrics

The full mapping of the ten SENAR §9 metrics to the REQ refinements from §12.3.

SENAR metric (§9)	REQ refinement from §12.3
Throughput	+ Coverage Velocity (§12.3.8) — at the requirements level
Lead Time	+ RDLT (§12.3.1) — for the requirements phase
FPSR (First-Pass Success Rate)	+ REQ-FPSR (share of artifacts passing QG-0 without rework) — derived, not a separate §12.3 metric
DER (Defect Escape Rate)	+ DRA (§12.3.5) — defects at acceptance

KCR (Knowledge Capture Rate)	(used as-is; indirectly reinforced through the §12.5.5 outcome)
Cost Predictability	+ variance of Cost per Approved Requirement (§12.3.9)
Cost-per-task	+ Cost per Approved Requirement (§12.3.9) — for the requirements phase
MIR (Memory Integrity Rate)	(used as-is; reinforced through V1 + V6 at RENAR-4+)
Cycle Time	+ RDLT (§12.3.1) + Requirement-to-Task Latency (§12.3.2) — both within SENAR Cycle Time
ADR (Adversarial Detection Rate)	+ ACR (§12.3.6) — adversarial for the requirements zone

Metrics with **no** SENAR counterpart (new in RENAR):

- Hallucination Rate (§12.3.3) — specific to AI-generated artifacts.
- Multi-model Disagreement Rate (§12.3.4) — specific to multi-model generation.
- Test-spec Drift Rate (§12.3.7) — specific to V5 requirement-version pinning.
- Reconciliation Findings per Week (§12.3.10) — specific to continuous reconciliation.

12.8 Relationship to other chapters

Chapter	Relationship
02 Positioning in the typology of methodologies	§2.3 Source-of-Truth inversion + §2.4.2 continuous reconciliation — the foundation for §12.3.10 Reconciliation Findings
07 ADAPT	§7.4.5 backward findings — input for §12.3.10; delta-ADAPT — measured through §12.3.5 DRA
08 Specifications	§8.5.7 SPEC-AI continuous evaluation — related to §12.3.4 Multi-model Disagreement Rate
09 Test cases	§9.12 last-run — input for §12.3.7 Drift Rate; §9.15 COVERAGE — the source for §12.3.8 Velocity and §12.3.7
10 Lifecycle and QG	§10.13 event audit log — the basis for all §12.3 metrics; §10.4.2 QG-4 — the gate at which §12.3.5 DRA is captured
03 Substrate versioning	V1 + V5 + V6 — capabilities mandatory for substrate-native collection of the §12.3 metrics (§12.6.1)
11 Maturity model	§12.3 targets per RENAR-3/4/5 level — a concretization of the level criteria from §11.4–§11.8
13 Conformance	§12.3 metrics — input for the §13.5 self-assessment; exceeding thresholds (for example, Hallucination Rate > 5% at RENAR-4) — a loss-of-conformance trigger §13.8.1

13. Conformance

> ***Part of the RENAR Standard v1.0*** · *← Table of contents*

> ***Dense chapter:*** *start with the reference/08 kit; the MVR↔§13.3 bijection is there too, in §1.*

13.1 How to prove conformance

Saying "we do everything the RENAR way" is easy — proving it is harder. This chapter is about how a project makes a verifiable claim: "we implement RENAR at the `RENAR-3` level" — and backs it up so that an external assessor can re-check it. The claim is not a slogan but a signed manifest with references to evidence in the substrate: here is the level, here is the evidence for each mandatory clause, here is who confirmed it and when.

The chapter normalizes three questions. **Who** is entitled to claim — the project Architect (self-assessment) or an independent assessor (third party). **On what basis** — the closed list of RENAR-1..5 levels plus the universal clauses mandatory for any level. **How the claim lives over time** — scheduled re-assessment, and upon a violation, loss of conformance. If the claim ceases to be true, this is a recorded event, not a silent omission.

The levels themselves are given here as a short semantic summary: the full RENAR-1..5 criteria are in chapter 11, and the substrate-native check mechanisms are in chapter 3 and `guide/06-compliance.md`. Here — the rules of the claim itself.

13.2 The closed list of RENAR levels

A closed list of five maturity levels. The full criteria are in chapter 11; below is the normative **semantic summary** for a conformance claim.

Level	Brief semantics	Conformance status
`RENAR-1`	Ad-hoc (spontaneous level): requirement artifacts are kept in a substrate with V1–V6 (§13.3.2); no formal `frontmatter` schema, no lifecycle statuses	Minimum entry level (§13.2.1)
`RENAR-2`	Documented (fixed in artifacts): a requirements substrate exists; artifacts have a basic `frontmatter`; the TZ is recorded	Conformance declarable (§13.2.2)
`RENAR-3`	Tracked (lifecycle and link accounting maintained): full `frontmatter` schema; lifecycle statuses are used; delta-TZ workflow through ADAPT	Conformance declarable (§13.2.2)
`RENAR-4`	Verified: 100% of `approved` artifacts have `verified-by`; pos/neg pairing (§9.7); QG-2 enforced; AI provenance	Conformance declarable (§13.2.2)
`RENAR-5`	Optimized: adversarial critic; multi-model generation; knowledge graph; Hallucination Rate measured	Conformance declarable (§13.2.2)

13.2.1 RENAR-1 as the minimum entry level

`RENAR-1` is the normative entry level. A project with no requirements substrate (where requirements live only in ticket systems or private correspondence) MUST NOT claim `RENAR-1`; a conformance claim against the standard begins with the actual presence of a requirements substrate.

`RENAR-1` is recorded in the conformance manifest only if the project explicitly declares the start of its RENAR journey; for projects with no intent to adopt RENAR, a conformance manifest is not required.

13.2.2 Closedness of the list

New levels (`RENAR-6`, `RENAR-N+`) MUST NOT be created locally at the project level. Changing the list of levels is possible only through the formal change procedure of the standard (§13.9).

An implementation MAY tighten requirements relative to the level (declare-stricter — see §10.10.2); this does not change the level and does not take the project outside the closed list.

13.3 Mandatory clauses, universal to all levels

Normative clauses mandatory for **any** conformance claim regardless of the declared level (including `RENAR-1`). Violating even one of them means the absence of conformance to the RENAR Standard as a whole.

13.3.1 Source-of-Truth inversion

An implementation MUST observe the **Source-of-Truth inversion** (§2.3): the requirement-artifact hierarchy is the source of truth about system behavior; code is a derived implementation artifact. Equivalent violations: reverse-engineering behavior from code into an SR without a bug-fix justification (§2.3.3 (1)); silent adaptation of an SR to the observed behavior of code (§2.3.3 (4)).

13.3.2 Substrate capabilities V1–V6

The project substrate MUST satisfy capabilities V1–V6 (chapter 3 §3.3):

Capability	Status for conformance
V1 — immutable history	**Mandatory absolutely**: without V1 an audit trail and V5 are impossible
V2 — atomic change unit	**Mandatory absolutely**: without V2 delta-ADAPT consistency is impossible
V3 — diff and review	**Mandatory absolutely**: without V3 there are no gates, no approval (§10.11.2)
V4 — branching / change-set	**Mandatory absolutely**: without V4 WIP and the source of truth are inseparable (§10.11.2)
V5 — cross-substrate version pin	**Mandatory absolutely**: without V5 `verifies[].version` and QG-2 are impossible (§10.3.3)
V6 — author and timestamp	**Mandatory absolutely**: without V6 the Architect's signature on an ADAPT, the dual signature of an ACTZ, and AI provenance are impossible (§10.11.2)

Negative scenario: a substrate in which at least one of the V1–V6 capabilities is missing normatively **cannot** implement any RENAR-N — including `RENAR-1`. This is a structural constraint (§3.2), not an operational one.

13.3.3 Reactive ADAPT

ADAPT is a reactive artifact (§7.4.1). Every TZ MUST pass an adversarial review (§7.10.2); its outcome MUST be issued as an AR — an adversarial-review record in status `issued` (V6 author + timestamp, §7.4.6). What follows depends on the verdict:

Adversarial reviewer's verdict	Requirement for ADAPT	Requirement for `source` of BR/SR/SPEC
"findings present" — backward findings, term mapping clarification, or scope clarification were detected	ADAPT is REQUIRED, in status `approved`, with the Architect's signature (§7.5). Every finding carries a `decided-in` pointing to a clause of a **signed** ACTZ (§7.13). Lifecycle §7.4.5. The AR carries a non-empty `produces-adapt[]`.	`source.adapt` mandatory; `source.tz-section` mandatory
"no findings, no clarifications" — the TZ → RENAR conversion is unambiguous	No ADAPT is created	`source.tz-section` mandatory; `source.adversarial-review-ref` mandatory — a reference to an AR with the `no-findings` verdict

Delegation to the adversarial reviewer is the only permissible way to declare "no ADAPT is needed." Creating BR / SR / SPEC from a TZ without a recorded verdict is a violation of the standard; hooks (§10.11.1 `adapt-applicability validation`) MUST block such artifacts.

In the presence of a delta-TZ — the same rule (§7.6): a delta-ADAPT is created upon findings from the adversarial review of the delta-TZ.

Multiplicity and stage independence. The ADAPT trigger is stage-agnostic: the verdict is rendered not only at TZ import but also at the derivation stages (BR → SR → SPEC → TC, §7.4.1.1). A single TZ has **zero or more** root ADAPTs (cardinality 0..N, MVR-3, §7.4.1.4); each records a `trigger-stage`. Multiplicity is conformant to the standard and does not weaken provenance: each BR / SR / SPEC references exactly one ADAPT or a `source.tz-section`.

Supersession (`supersession`). An approved/frozen ADAPT MAY be superseded by a superseding ADAPT (§7.6.4). A conformant supersession MUST: (1) preserve the superseded ADAPT in the terminal `superseded` state (immutable, V1) — not delete it; (2) upon a contractual outcome (the decision was a clause of a signed ACTZ), be formalized by a **new ACTZ with the dual signature**, which stamps `superseded-by` on the former one; (3) redirect or re-derive all derivatives so that no dangling `source.adapt` pointing at a `superseded` one remains (§6.10.3). A separate QG is not required — QG-3 is used (§10.8.5).

Negative scenarios (non-conformant):

- The manifest declares conformance, but BR/SR/SPEC are derived from a TZ without `source.tz-section` and without `source.adapt` — a violation of mandatory provenance.

- Creating BR/SR/SPEC with `source.adapt` omitted **without** the evidence `source.adversarial-review-ref` — a violation of §7.4.1.3 "no silent skip."
- `source.adversarial-review-ref` points at a non-existent AR, at an AR in status `draft` (the verdict is not issued), or at an AR in status `superseded` (the verdict was displaced) — invalid evidence (§7.4.6).
- An AR issued with a reviewer model equal to the primary agent's model — the review is not adversarial (§7.10.2).
- An AR with `verdict: findings-present` and an empty `produces-adapt[]` — the verdict claims findings, yet no ADAPT was produced.
- A finding in status `resolved` without a `decided-in` pointing to a signed ACTZ — the interpretation rests on a decision the client never took (§7.4.5).
- A signed ACTZ none of whose decisions is reflected in an ADAPT — an obligation exists outside the requirements.
- The client signature on an ADAPT declared mandatory — a redundant requirement; it is permissible only as `declared-stricter` (§1.7.2), not as baseline conformance.
- **An AT generated by an agent that had access to the internal contour** (ADAPT / BR / SR / SPEC / TC / code) — the acceptance level collapses into the verification level, and an interpretation error receives a false confirmation of conformance (§9.19.2).
- The product submitted for trials with AT derived from an **outdated** revision of the effective TZ (§9.19.4).
- A TC counted as evidence **without a red history** and without a killed mutant (§9.18.2).
- A requirement of the `implementation-originated` class describing **client-observable** behavior (§6.13.2) — an obligation toward the client cannot originate in the code.
- An ADAPT created under a "no findings" verdict (evidence exists) — a contradiction: the verdict claims that no ADAPT is needed, yet an ADAPT is present.
- An ADAPT in status `superseded` deleted from the substrate — a violation of immutable history (V1).
- A dangling `source.adapt` reference to a `superseded` ADAPT (the derivative was not redirected or re-derived) — an invalid trace chain.
- Supersession of a decision with a contractual outcome without a new signed ACTZ — a unilateral cancellation of what was agreed with the client, a violation of §7.6.4.

13.3.4 Closed list of 11 SPEC types

A SPEC type MUST belong to the closed list of eleven types (§8.3): `SPEC-ARCH`, `SPEC-API`, `SPEC-DATA`, `SPEC-INT`, `SPEC-PROC`, `SPEC-UI`, `SPEC-AI`, `SPEC-SEC`, `SPEC-OPS`, `SPEC-TEST`, `SPEC-DOC`.

A project MUST NOT create new SPEC types locally. Changing the list is possible only through the formal change procedure of the standard (§8.3.1, §13.9).

13.3.5 TC pos/neg pairing for normative statements

Every normative statement of a verifiable artifact (BR / SR / SPEC / TR) that is covered by at least one TC MUST have a paired negative TC (§9.7). Single-TC coverage is permitted only in one case: the statement itself describes a negative invariant (for example, a `SPEC-SEC` STRIDE category).

QG-2 (§10.3.3) MUST block promoting an artifact to `verified` in the absence of at least one paired negative TC.

13.3.6 Closed list of Quality Gates

The closed list of Quality Gates (§10.3, §10.4):

Gate	Status in the conformance manifest
QG-0 — Approval	**Mandatory required**
QG-1 — Implementation	**Mandatory required**
QG-2 — Verification	**Mandatory required**
QG-3 — Architecture (opt.)	`declared` (supported) or `absent`; for projects with mandatory ADAPT — effectively always `declared`, since ADAPT approval operates with the Architect's signature at QG-3 (§10.4.1)
QG-4 — Acceptance (opt.)	`declared` or `absent`; when `absent`, the artifact's terminal status is `verified` (without `accepted`)

Creating new gate types locally is prohibited (§10.10.2). Locally tightening the preconditions of a canonical gate is permitted (declare-stricter); locally weakening them is prohibited.

13.3.7 Closed lists of backward findings and SPEC decompositions

- The list of backward-findings categories in ADAPT is closed (§7.4.4) — 7 categories; adding new ones is the formal change procedure of the standard (§13.9).
- The closed list of SPEC types is additionally fixed by §13.3.4.
- The closed list of artifact lifecycle states (chapter 10) — is not extended locally at the project level.

13.3.8 Cross-level subsystem traceability (`implements`-edge)

For the "subsystem as a standalone product" scenario (§6.8.2), an implementation MUST provide a machine-readable trace chain `BR (subsystem) → BR (system)` through the typed `implements[]` edge (§6.5.2):

Conformance level	Requirement
v1.0: `recommended`	If `BR.level = subsystem` AND the parent system has ≥1 approved BR — the presence of `implements[]` is RECOMMENDED. Its absence is permitted but REQUIRES a justification in the `Context` section with a reference to ADAPT§.
v1.1+: `mandatory`	The same trigger makes `implements[]` mandatory. Its absence when the trigger fires is non-conformance.

In both versions the substrate MUST implement the `implements`-edge validation checkpoint from §10.11.1 (the target exists and is `approved`+, no cycles, deprecated target → warning, `implements[]` not on `level: system`).

Negative scenario: a project declares conformance to `RENAR-N` with `BR.level = subsystem` and an approved parent BR in the same system, but without `implements[]` and without a justification in the Context — non-conformant from v1.1 onward; on v1.0 — predictably-non-conformant upon upgrade (§13.9 migration guidance).

13.4 The conformance manifest

13.4.1 Location and format

The conformance manifest is stored at the root of the project's requirements substrate under the name `RENAR-CONFORMANCE.yaml`. The format is YAML 1.2; an alternative serialization (`.json`) is permitted as an **additional** artifact, not a replacement.

The manifest is **immutable** in the V1 sense: each conformance claim creates a new version of the manifest (`manifest-version` is incremented); previous versions are not deleted, they remain in the substrate as an audit trail. Replacing the manifest through `replaced-by` points to the new version.

13.4.2 Mandatory fields

```yaml
# RENAR Conformance Manifest schema (mandatory fields, v1.0)
renar-version: "1.0"           # the version of the RENAR Standard against which
conformance is claimed
senar-version: "1.0"           # the version of SENAR the implementation relies
on (mandatory, MVR-7)
manifest-version: 3            # incremented on each update; not re-used
manifest-id: "CFM-2026-001"    # stable substrate-native identifier (V1)

# Conformance level
level: "RENAR-3"               # from the closed list §13.2 (RENAR-1..RENAR-5)
level-target: "RENAR-4"        # optional: the next target level (for path
planning)

# Assessment metadata
assessment-mode: "self"        # self | third-party
assessment-date: "2026-05-13"  # ISO-8601 date of completion of the current
assessment
assessor:
  id: "architect-andrey-y"     # V6 author identifier
  role: "architect"            # role from §13.5 (architect | authorized-role-
holder | external-assessor)
  signature-ref: "<substrate-native pointer to the signature event>"
next-assessment-due: "2026-08-13"  # §13.7

# Mandatory clauses confirmation (§13.3)
mandatory-clauses-confirmed:
  sot-inversion: true                              # §13.3.1
  substrate-v1-v6: { v1: true, v2: true, v3: true, v4: true, v5: true, v6: true }
# §13.3.2
  adapt-per-tz: true                               # §13.3.3
```

```yaml
  spec-types-closed-list: true                    # §13.3.4
  tc-pos-neg-pairing: true                        # §13.3.5
  quality-gates-closed-list: true                 # §13.3.6
  closed-lists-backward-findings: true            # §13.3.7

# Quality gates declaration (§10.4.4)
quality-gates:
  qg-0: required              # required (mandatory)
  qg-1: required
  qg-2: required
  qg-3: declared              # required | declared | absent
  qg-4: absent

# Substrate capabilities declaration (§13.3.2)
substrate-capabilities:
  v1-immutable-history: declared
  v2-atomic-change-unit: declared
  v3-diff-review: declared
  v4-branching: declared
  v5-version-pin: declared
  v6-author-timestamp: declared
  substrate-id: "<substrate-native pointer to guide/03..06>"   # cross-ref to
guide

# Spec types support (§13.3.4)
spec-types-supported: ["SPEC-ARCH", "SPEC-API", "SPEC-DATA", "SPEC-INT",
                       "SPEC-PROC", "SPEC-UI", "SPEC-AI", "SPEC-SEC", "SPEC-OPS",
                       "SPEC-TEST", "SPEC-DOC"]
# All 11 types are mandatory as the minimum-supported; a declaration "type X is
not used in the project" is permitted,
# but it CANNOT be a declaration "type X is not supported substrate-natively."

# Optional fields
declared-stricter:            # §13.2.2, §10.10.2 — local tightening
  - clause: "QG-2"
    override: "required-negative-tc-per-clause"
    rationale: "regulated industry, double safety margin"
  - clause: "tc-pos-neg-pairing"
    override: "100% (no single-TC exception for security)"
    rationale: "mandatory requirement of the internal ISMS"

exceptions: []                # declared exceptions relative to the base level
(with justification in the audit trail)

# External conformance records per §14.4 (optional; for the value field see the
`claim` key below — substrate §14.6)
external-claims:
  - standard: "ISO/IEC 5338:2023"
    clause-ref: "§2.4.x"
    claim: "partial"          # full | partial | aligned
  - standard: "NIST AI RMF 1.0"
    clause-ref: "§2.4.x"
    claim: "aligned"
```

```
replaced-by: null              # substrate-native pointer to the next version of
the manifest, if released
replaces: "CFM-2026-001@v2"    # substrate-native pointer to the previous version
of the manifest
```

13.4.3 Field semantics

- `renar-version` — the version of the standard; conformance is claimed against a specific point of the standard; upon the release of a minor version of the standard (§13.9), a re-assessment is REQUIRED (§13.7) with an update of `renar-version`.
- `senar-version` — the version of SENAR the implementation relies on; mandatory per MVR-7 (§0.5); its absence from the manifest is non-conformance (§13.8).
- `manifest-id` — V1 immutable identifier; not re-used even after `replaced-by`.
- `level` — the closed list §13.2; a violation of the mandatory clauses (§13.3) means conformance is absent regardless of `level`.
- `level-target` — an optional declaration of the development path; it is not a normative obligation.
- `assessment-mode` — `self` (§13.5) or `third-party` (§13.6); `third-party` MUST contain a formal reference to the external assessment act.
- `assessor` — V6 author + role; for `third-party` — an external participant with explicit identification.
- `next-assessment-due` — `assessment-date + cadence` (§13.7); an overrun is a trigger for loss of conformance (§13.8).
- `quality-gates` — MUST contain all five gate-ids; the status of `qg-0..qg-2` ∈ {required}; `qg-3, qg-4` ∈ {required, declared, absent}.
- `mandatory-clauses-confirmed` — each field set to `true` is mandatory for any conformance claim; `false` makes the manifest invalid.
- `declared-stricter` — a list of local tightenings; MUST contain `clause`, `override`, `rationale`.
- `exceptions` — a list of declared exceptions relative to the base level; each exception REQUIRES a justification in the audit trail and MUST NOT touch the mandatory clauses.
- `external-claims` — an optional list of conformance records against external normative references (§14.4, §14.6); each record contains `standard`, `clause-ref`, and the field `claim` (value: full | partial | aligned) — a technical schema key, with no term substitution in the manifest.

13.5 Self-assessment procedure

13.5.1 Actor

Self-assessment is conducted by the project **Architect** or by an **authorized role holder** (chapter 5) explicitly declared responsible for conformance.

13.5.2 Methodology

Steps: (1) the §13.3 checklist — the actor checks each mandatory clause; the result goes into `mandatory-clauses-confirmed`; at least one `false` → the assessment is not completed, and a remediation plan for the violations is formed; (2) the chapter 11 §§11.4-11.8 checklist for the declared `level` — each level criterion = a separate check with evidence in the substrate; (3) filling in the manifest (all mandatory fields §13.4.2; `level` not higher than the passed checklist); (4) signing and publishing — the actor signs the manifest with the native mechanism (V6 author + timestamp); the manifest becomes part of the source of truth through V3 review (for self-assessment, self-approval is permitted, the evidence MUST be recorded).

13.5.3 Evidence

Each mandatory-clause check MUST be accompanied by references to evidence in `audit-trail/CFM-<id>/<clause>.md` (V1 — a stable identifier pointing to specific artifacts, runs, events). Evidence is retained indefinitely (V1 + §10.13.3 retention).

13.5.4 Cadence

The first claim (kickoff); the cadence is §13.7; the trigger is §13.8 (loss of conformance); the release of a minor version of the standard (§13.9).

13.6 Third-party assessment (optional)

An optional path to confirming conformance by an external assessor. It applies in regulatory contexts (medicine, fintech, the public sector, AI-critical systems) or under contractual obligations toward the client.

13.6.1 Actor

The external assessor is an independent participant with read-only access to the substrate for the assessment period. The formal qualification is substrate-specific and is recorded in `assessor.signature-ref` (an external auditor of a certified organization).

13.6.2 Methodology

The same as §13.5.2 (the §13.3 checklist + the chapter 11 checklist), plus: an audit trail — all of the assessor's actions (read-events) are recorded natively; independent verification — the assessor checks the evidence independently without relying on the self-assessment results; the outcome — a signed manifest (the assessor signs with the native V6 mechanism) **or** a denial with a justification and a list of specific violations.

13.6.3 Additional manifest fields

When `assessment-mode: third-party`, the following are mandatory:

```
third-party:
  assessor-organisation: "<name>"
```

```
  assessor-qualification-ref: "<pointer to the qualification document>"
  audit-report-ref: "<pointer to the full audit report>"
  audit-log-ref: "<pointer to the read-events log>"
```

13.6.4 Relationship between self / third-party

Third-party **does not cancel** the self-assessment cadence (§13.7); the paths are compatible. A project MAY run self-assessment quarterly and third-party annually; the manifest is versioned separately for each path with different `manifest-id` s.

13.7 Re-assessment cadence

13.7.1 Default

Self-assessment is **quarterly** (every 3 months from `assessment-date`); third-party is **annual**. It is declared in the manifest's `next-assessment-due` field.

13.7.2 Override

The cadence MAY be overridden in `declared-stricter` :

```
declared-stricter:
  - clause: "re-assessment-cadence"
    override: "monthly"
    rationale: "AI-critical project, eval dependency"
```

Weakening the cadence (`override: "annual"` for self under the default `quarterly`) is **prohibited** — a violation of §13.3.1 Source-of-Truth inversion (a hidden weakening = concealing a loss-of-conformance event).

13.7.3 Immediate re-assessment triggers

The release of a minor version of the RENAR Standard (`renar-version` shifts); a violation of a mandatory clause (§13.3, the loss-of-conformance trigger §13.8); a substantial change of substrate (substrate replacement — V1-V6 is re-assessed); a substantial change of scope (a new artifact class, a new SPEC type).

13.8 Loss of conformance

13.8.1 Triggers

Conformance is considered **lost** upon the occurrence of any of:

Trigger	Description
A mandatory clause is violated	Any of §13.3.1–§13.3.7 ceased to hold (for example, a BR appeared without `source.adapt` and without `source.adversarial-review-ref` — a silent skip of the adversarial review; the substrate lost V3 after migration)
Substrate capability degradation	V1, V2, V3, V4, V5, or V6 is no longer provided by the substrate (for example, migration to a substrate without diff & review)
The manifest expired	`next-assessment-due` passed without releasing a new version of the manifest
Level criteria violated	The criteria of the declared level (see chapter 11) ceased to hold without a formal downgrade
Metric threshold exceeded	A critical metric of chapter 12 exceeded the normative threshold for the level (for example, Hallucination Rate > 5% on RENAR-4 — §12.3.3 — an explicit normative trigger)
External denial	An external assessor (third-party assessor) returned a denial with justification

13.8.2 Procedure

Steps: (1) **recording the loss-event** in the audit trail (`audit-trail/CFM-<id>/loss-events/<timestamp>.md`); (2) **a formal downgrade or a declaration of unknown-state** — a downgrade (a new version of the manifest with a `level` below the current one, if the mandatory clauses hold) or unknown-state (an explicit declaration in public communications; the manifest is marked `replaced-by: "<unknown-state>"` with a sentinel value, distinct from the default `null` ; re-assessment is REQUIRED for recovery); (3) **a recovery plan** — `recovery/CFM-<id>-<timestamp>.md` specifying the deadline and the mandatory clauses / level criteria; (4) **re-assessment** after the recovery plan — self-assessment §13.5 with an updated manifest is REQUIRED; for third-party — a repeated external assessment.

13.8.3 Public communications

Loss of conformance, if the level is claimed publicly, MUST be recorded publicly within a reasonable time (the notification cadence — `guide/`). Concealing the fact of loss is a violation of §13.3.1 (the Source-of-Truth audit trail).

13.9 The closed-list policy for RENAR-N levels

13.9.1 Normative rule

The closed list of levels RENAR-1..RENAR-5 (§13.2) is **not extended** locally at the project level. Any project claiming conformance MUST specify `level` ∈ {RENAR-1, RENAR-2, RENAR-3, RENAR-4, RENAR-5}. This policy is a specialization of the §1.7 closed-list policy for maturity levels; the master index is §1.7.5.

13.9.2 What is prohibited

Action	Prohibited?	Why
Locally creating a level at the project level (`RENAR-6` , `RENAR-PRO`)	Prohibited	Violates the closed list; conformance is non-portable
Locally overriding criteria (weakening RENAR-4 without a formal downgrade)	Prohibited	Violates the contract of the standard
Locally tightening criteria (declare-stricter)	Permitted	See §13.4.2 `declared-stricter`
Claiming a level higher than the one actually achieved	Prohibited	Violates §13.3.1 Source-of-Truth inversion
A conformance claim without a manifest	Prohibited	§13.4 mandatory
Intermediate levels such as `RENAR-3.5`	Prohibited	The list is discrete

13.9.3 The path to adding a new level

Only through the formal change procedure of the standard: a research draft (justification, a typology of criteria, a comparison with existing RENAR-N) → public review → a minor- or major-version bump of the standard → migration guidance (for existing conformant projects: changes to the self-assessment checklist, new manifest fields).

Project-local extensions remain outside conformance — permitted as internal practices (`declared-stricter`), but they do **not** affect the formal `level` in the manifest.

13.10 Relationship to other chapters

Chapter	Relationship
02 Positioning in the typology of methodologies	§2.3 Source-of-Truth inversion — the mandatory clause §13.3.1; §2.7 the conformance consequence explicitly refers back to this chapter
06 Requirements hierarchy	the `frontmatter` of artifacts (BR / SR / TR) — the RENAR-2/-3 level criteria are checked per §6.5–§6.7 (chapter 11)
07 ADAPT	the mandatory clause §13.3.3 (reactive ADAPT: an adversarial review for each TZ, ADAPT per the verdict); §7.4.4 the closed list of backward findings — §13.3.7
08 Specifications	the mandatory clause §13.3.4 (the closed list of 11 SPEC types); §8.3.1 the closed-list policy — §13.9
09 Test cases	the mandatory clause §13.3.5 (pos/neg pairing); §9.10 QG-2 — §13.3.6
10 Lifecycle and QG	§10.3 the canonical gates, §10.4.4 the conformance-manifest fragment — expanded here into the full manifest schema; §10.10 the closed-list policy for gates parallels §13.9 for levels
03 Substrate versioning	the mandatory clause §13.3.2 (V1–V6 declared); §3.4 the mapping table — non-normative, informational

11 Maturity model	detailed criteria for levels RENAR-1..5 — here §13.2 contains a semantic summary; the full per-level checklist is in §§11.4–11.8 (one section per level)
12 Metrics	the metrics on which self-assessment is built (`approved-without-verified` , `pos-neg-pairing-percent` , and others) — specified here as input data for §13.5

14. Normative references

> ***Part of the RENAR Standard v1.0*** · *← Table of contents*

14.1 Which standards RENAR builds on

An engineer who opens this chapter has one practical question: which of the listed standards must I actually comply with, and which are just background? RENAR answers directly. It does not rewrite other people's standards — it builds on them and claims conformance only in part, not in whole. Hence the split: **normative references** (§14.4) — what RENAR actually declares conformance to (ISO/IEC/IEEE 29148, ISO/IEC 5338, and others); **informative references** (§14.5) — methodologies and terminology used for positioning, not required for a conformance claim; **conformance positioning** (§14.3) — a single formulation that places RENAR among related standards; **what RENAR does not adopt** (§14.7) — a closed list of non-borrowed practices. The full extended catalog of informative mappings — reference/11.

If a normative reference conflicts with a clause of this chapter, the clause of this chapter prevails (RENAR is a specialization and adaptation). RENAR makes a conformance claim against an external standard **only** in the stated part, not in whole. All dates are the publication date of the referenced edition; the reference is **dated** (§14.4.1).

14.2 SENAR as the parent standard

RENAR is a **specialization of SENAR** (§1.6.1) in the requirements-engineering domain. RENAR does not duplicate the following SENAR clauses; they apply as-is:

SENAR clause	Used without rewriting
5 values and 14 rules	Context for all of RENAR; see §1.6.1
QG-0 / QG-1 / QG-2 as a concept	RENAR makes the state machines concrete in §10
10 common process metrics	RENAR adds 11 domain metrics in §12.3
5 levels of general maturity	RENAR-M is a separate dimension (§11.2)
5 roles (Supervisor, AI agent, Architect / Tech Lead, Reviewer, Stakeholder)	RENAR does not redefine the roles; see §5
Agent instrumentation (control levels, profiles)	RENAR extends it for requirements specifics

RENAR begins where SENAR ends: SENAR is the general methodology of AI-native development; RENAR is the normative requirements-management document for SENAR-compatible systems.

14.3 Conformance positioning

> ***RENAR — requirements management aligned with ISO/IEC/IEEE 29148:2018, adapted for development with AI agents, built on top of the SENAR methodology, and compatible with SAFe 6.0 coordination.***

This formulation is the point of reference for all conformance claims that projects issue based on RENAR (§13.4).

14.4 Normative dated references

Each entry in §14.4.2–§14.4.6 contains the blocks **"What it normalizes"**, **"How RENAR relates"**, and **"Conformance claim"**. The blocks **"What RENAR adapts"** and **"What RENAR does not adopt"** appear only in deep-adaptation entries (29148 and 5338); the other entries are shorter — this reflects the depth of the claim, not a structural defect.

14.4.1 The notion of a dated reference

RENAR uses **dated references**: each normative reference cites a specific edition of a standard (with its year). Undated references do not apply — semantics change between editions, and a conformance claim MUST be verifiable against a pinned edition.

Lifecycle of a normative reference. *Active* — the reference is cited in the current version of RENAR (`renar-version` in the conformance manifest, §13.4.2). *Updated* — the referenced standard has released a new edition; RENAR is updated within a reasonable timeframe (the project manifest pins a `renar-version`, which in turn pins the editions of external references). *Withdrawn upstream* — the referenced standard has been retired (like IEEE 830-1998); RENAR moves the reference to §14.5 and names the successor.

Triggers for immediate re-evaluation. When a new edition of one of the references in §14.4 is released, an immediate re-evaluation (§13.7.3) of project manifests is REQUIRED: checking the RENAR chapters for consistency (a changelog entry); updating `renar-version` in project manifests; an entry in the substrate audit-trail.

Negative scenario: a RENAR conformance claim of `renar-version: 1.0`, when a new edition of ISO/IEC 5338 is released, **without** updating `renar-version` in the manifest — is invalid; the substrate hook (§13.8.1) detects the stale version.

14.4.2 ISO/IEC/IEEE 29148:2018 (requirements engineering)

Official title (EN): Systems and software engineering — Life cycle processes — Requirements engineering.

What it normalizes: the international requirements-engineering standard — stakeholder needs, specification, validation, verification, attributes, traceability, lifecycle.

How RENAR relates:

29148	RENAR	Type
Requirement classes: stakeholder, system, software	BR, SR, TR	Borrows with renaming
Requirement attributes (18 in 29148)	Mandatory minimum in frontmatter (§6.5.2, §6.6.2) — 7–8 fields	Simplifies

SRS structure	The requirements substrate structure is isomorphic	Borrows
Verification methods: inspection, analysis, demonstration, test	TC (§9) — a full-fledged artifact; inspection — via the `[test-spec-change]` workflow (§9.13)	Adapts

What RENAR adapts:

- 29148 provides for 18 attributes; RENAR keeps 7–8 mandatory ones, the rest being auto-derived (§4.12) or optional. Rationale: in development with AI agents, excessive attribution increases the risk of hallucinations (§12.3.3).
- 29148 does not single out TC as a separate artifact. RENAR makes TC a full-fledged artifact (§9).

What RENAR does not adopt: the formal review meetings and walkthroughs of 29148 — replaced by QG-0 / QG-2 and adversarial AI review (§11.7 for RENAR-4+).

Conformance claim: RENAR claims conformance with ISO/IEC/IEEE 29148:2018 in the part covering requirement classes, attributes, lifecycle, and verification methods (pinned in the manifest, §13.4.2).

14.4.3 ISO/IEC 25010:2023 (product quality model)

Official title (EN): SQuaRE — Product quality model.

What it normalizes: nine software quality characteristics (the 2023 edition), including the new Safety.

How RENAR relates: the 25010 characteristics are **mandatory categories** for non-functional SR. The frontmatter of a non-functional SR (§6.6.2) MUST contain a `quality-characteristic` from the 25010 list; for a functional SR the field is omitted.

Conformance claim: RENAR claims conformance with ISO/IEC 25010:2023 in the part covering the category vocabulary for NFRs.

14.4.4 ISO/IEC 25022:2016 / 25023:2016 (quality measures)

What it normalizes: formal measures for each 25010 characteristic (for example, response time in ms).

How RENAR relates: the Pass criteria in TC (§9.11.1) MUST be expressed through 25022/25023 measures where applicable. Example: "p95 < 200 ms at 100 RPS" instead of "performance is acceptable".

Conformance claim: RENAR claims conformance in the part covering measurable Pass criteria for TC.

14.4.5 ISO/IEC 5338:2023 (AI-system lifecycle)

What it normalizes: the first international standard for the AI-system lifecycle (an adaptation of ISO/IEC 12207).

How RENAR relates:

- **Decision logs** — material decisions by the AI agent are documented; implementation: audit-trail (§10.13) + `ai-provenance` (§4.10.1).
- **Data versioning** — eval-datasets with version pin V5 (§3.3.5).
- **Model versioning** — `ai-provenance.generated-by` is mandatory at RENAR-4+ (§11.7.1).
- **Continuous validation** — scheduled eval-runs (§11.8.1 for RENAR-5).

What RENAR adapts:

- 5338 describes the full lifecycle of an AI system (derived from ISO/IEC 12207); RENAR borrows from it only the processes concerning requirements and the provenance of AI-generated artifacts — decision logs, data and model versioning, continuous validation — and expresses them through `ai-provenance` (§4.10.1) and the RENAR-4 / RENAR-5 levels (§11.7, §11.8).

What RENAR does not adopt: the operational layer of the AI-system lifecycle — model training, deployment, and operation — is outside the scope of RENAR (§1.3); the standard normalizes only the requirements axis and the provenance of AI-generated artifacts (`ai-provenance`).

Conformance claim: RENAR claims conformance in the part covering the AI-artifact lifecycle and AI-assisted artifact generation.

Negative scenario: a conformance claim against 5338 without `ai-provenance` (§4.10.1) — is invalid. RENAR MUST refuse to issue a manifest for such projects.

14.4.6 ISO/IEC 23894:2023 (AI risk management)

What it normalizes: AI risk classes and mitigation strategies.

How RENAR relates:

Risk (23894)	Mitigation in RENAR
Hallucinations in AI output	Source citation (§4.10.2), the Hallucination Rate metric (§12.3.3)
Model drift	pinning `last-run.requirement-version` (§9.12) + periodic re-run
Prompt injection via input data	Sanitization on TZ import (substrate-specific, in guide/)
Bias in AI generation	Multi-model agreement for critical BR (RENAR-5, §11.8.1)
Adversarial inputs	Adversarial review (§11.7.1, §11.8.3)
Single point of failure (one model)	Multi-model agreement; isolation of the judge model (§9.13.4)

Conformance claim: RENAR claims conformance in the part covering identification and mitigation of AI risks in the requirements domain; the full register — reference/03.

14.5 Informative references

Informative references are methodologies and terminology used for positioning; RENAR does **not** make a conformance claim against them.

14.5.1 The five key ones (start here)

Source	Why for RENAR
SAFe 6.0	Mapping of the Epic/Feature/Story hierarchy → BR/SR/TR (§4.13.1)
Spec-Driven Development	Source-of-Truth inversion (§2.3.1) as a formal paradigm
EARS (Mavin et al.)	Phrasing templates for SR and TC (§6.6.3)

BDD / Gherkin / Specification by Example	Prior art for a full-fledged TC (§9)
NIST AI RMF 1.0	Functional mapping of Govern/Map/Measure/Manage

Brief explanations — in reference/11 §1–§2.

14.5.2 Extended catalog

Source	Role for RENAR
IEEE 830-1998 (deprecated)	Historical reference; the normative successor — §14.4.2
BABOK v3	BA terminology; the elicitation gap — in guide/
PMBOK 7	Principles instead of processes (§2.5)
ISTQB Foundation	Testing vocabulary, compatible with `tc-type`
CMMI v2.0	Prior art for maturity levels (§11)
ISO/IEC 42001:2023	Organizational governance over RENAR
ISO/IEC 25059:2023	AI-system quality (an extension of 25010)
EU AI Act (Reg. 2024/1689)	The `ai-act.risk-class` field; legal conformance — outside RENAR
SysML / MBSE	Prior art for "requirements as a graph" (reference/05)

Detailed mappings — reference/11.

14.6 Conformance summary

14.6.1 Summary table

Standard	Type	Level	RENAR chapters
SENAR	Parent	Specialization	All
ISO/IEC/IEEE 29148:2018	Normative	High	06, 09
ISO/IEC 25010:2023	Normative	Medium	06, 08
ISO/IEC 25022/25023	Normative	Medium	09
ISO/IEC 5338:2023	Normative	High	04 §4.10, 11
ISO/IEC 23894:2023	Normative	Medium	11, reference/03
The rest (§14.5)	Informative	—	see reference/11

14.6.2 Aggregate claims

A manifest (§13.4.2) MAY contain **several** conformance claims against the references in §14.4 at the same time. Each is verified independently. A partial claim is not provided for: a project is either conformant through RENAR, or it is not.

Mandatory clauses and external standards. The mandatory clauses of §13.3 are RENAR's internal requirements. The claims of §14.4 are external, optional above the minimum (for example, RENAR-1 without a claim against 5338, if there is no AI generation of artifacts).

14.7 What RENAR fundamentally does not adopt

A closed list of practices from related standards:

Practice	Source	Why it is not adopted
Heavy document review meetings, IEEE 1028 inspections	RUP, SWEBOK	Incompatible with AI-agent speed; replaced by adversarial AI review (§11.7.1)
Manual verification only (29148 inspection meetings)	ISO/IEC/IEEE 29148 §6.4	Substrate hooks (§10.11) + AI review
Process-first CMMI (CCB, OSP)	CMMI v2.0	Principles + automated enforcement (§2.5)
Formal methods for all requirements (B, Z, TLA+)	Formal methods	Critical safety domains only, not the base level
Undated references to standards	Industry practice	Dated only (§14.4.1)
Self-declared conformance without a manifest	Industry practice	A manifest is mandatory (§13.4)

14.8 Relationship to other chapters

Chapter	Relationship
04 Terms	§4.13 — detailed mapping to related standards
02 Methodology	§2.3.4 SDD; §2.6 implications for mandatory clauses
06 Hierarchy	frontmatter — implementation of the 29148 attributes
09 Test cases	TC — an extension of 29148; ISTQB-compatible terminology
10 Lifecycle and QG	QG — concretization of SENAR QG-0..2
11 Maturity	The RENAR-1..5 levels
13 Conformance	`renar-version`, `external-claims[]`
reference/11	Full informative catalog of §14.5
reference/03 AI risk	Risk register per 23894 + NIST

00. Quickstart

> *End-to-end example: a small "email/password sign-up" project. The full RENAR cycle with a minimal set of artifacts: TZ → ADAPT → BR → SR → SPEC → TC. Time: ~30 minutes of reading + sketching on paper; ~2-3 hours if you do it live on a substrate. Prerequisites: core/renar-core.md (≤ 10 min). The full-size example with 2FA — 01-walkthrough.md. The RU corpus language — standard/00 §0.7.*

After this start you will have: an understanding of the full RENAR cycle on one small example; ready-made YAML templates for each artifact type; experience passing two gates (QG-ADAPT-approve ≡ **QG-3 Architecture Gate** §10.4.1 + **QG-2 Verification Gate**); a point from which to scale.

Prerequisites

Substrate (`substrate`) — the system for storing and versioning artifacts. RENAR is independent of the kind of store; for the quickstart, any substrate with V1-V6 capabilities will do (reference/01 §2.7): git with PR review; a document-oriented store; a DBMS with history and signatures.

Folder structure (the default convention for git):

```
my-project.req/
  tz/                    # immutable client TZ
  actz/                  # ACTZ — TZ clarification protocols (dual signature)
  adapt/                 # ADAPT artifacts
  br/                    # Business Requirements
  sr/                    # System Requirements
  specs/{arch,api,data,ui,sec,ai,proc,int,ops,test,doc}/
  tests/                 # TC (tc-type incl. contract)
  at/                    # AT — acceptance tests derived from the effective TZ
```

For other substrates, use an equivalent organization by document type or namespace.

Step 1. TZ (5 min)

The client provides a small TZ. It is a **contractual immutable** document — once signed, it is not edited.

`tz/TZ-2026-001.md` (excerpt):

```
---
id: TZ-2026-001
title: "User registration via email"
signed-date: "2026-05-15"
signed-by-client: "J. Ivanov, PM, ClientCo"
---

# TZ-2026-001
```

```
## §1. Context
Build a system for user registration via email.

## §2. Requirements
- A user can register via email and password.
- After registration, the user confirms their email.
- After confirmation — access to the personal dashboard.

## §3. Constraints
- Web application only.
- Storage in the RF (ФЗ-152).
```

The TZ is signed by the client → immutable. All edits and interpretations go through ADAPT.

Step 2. ADAPT `draft` (10 min)

Create `adapt/ADAPT-001-main.md`. An AI agent or an engineer fills in **Forward** (how we understood it) and **Backward** (what is unclear).

```
---
id: ADAPT-001
title: "Adaptation of TZ-2026-001 — Registration via email"
type: ADAPT
source-tz: { id: TZ-2026-001, signed-date: "2026-05-15", signed-by-client: "J.
Ivanov, PM, ClientCo", document-version-ref: "<version id>" }
status: draft
created: "2026-05-16"
ai-provenance: { generated-by: "anthropic-claude-opus-4-7@2026-05-16", prompt-
template: "prompts/adapt-from-tz.md@v1.0", context-tokens: 1024, output-tokens:
2048, human-edits: false }
---
```

Forward §2 Requirements:

```
**Quote:** "A user can register via email and password. After registration, the
user confirms their email. After confirmation — access to the personal
dashboard."

**Interpretation:** POST /auth/sign-up with {email, password}. A User is created
with status `unverified`. A verification link email is sent. Click → status
`verified`. Only `verified` users can sign in.

**Elaborated scenarios:** sign-up; email verification; first sign-in.
**Coverage:** in scope — email/password + verification + dashboard access. Out of
scope: OAuth, SMS, 2FA, password reset.
**Forward links** (auto-populated after approval): BR-01, SR-01, SR-02, SR-03.
```

Backward (3 entries):

```
### B-001: gap — sign-in attempt by an unverified user
Status: open. Description: the TZ does not describe system behavior on a sign-in
attempt before email verification.
Question to client: show a "verify your email" message with a resend button — or
a 401 with no explanation?

### B-002: regulatory — Ф3-152 storage in the RF
Status: open. Description: TZ §3 requires "storage in the RF". What exactly: the
data center, the provider's jurisdiction, a separate installation?
Question to client: clarify the scope of Ф3-152.

### B-003: gap — resending the email
Status: open. Description: what if the verification email is lost? what about a
rate limit?
Question to client: the resend policy.
```

Step 3. ACTZ and ADAPT `approved` (5 min)

Questions to the client do not go "into the email thread": the Architect collects them into an **ACTZ** — an agreed clarification of the TZ (in contractual usage, "TZ Clarification Protocol No. 1"). An ACTZ is signed by both parties, and it is the ACTZ that carries contractual weight (standard/07 §7.13). The backward lifecycle: `open → asked-to-client → answered → resolved → frozen (after approval)`; the protocol lifecycle: `draft → sent → signed`.

`actz/ACTZ-001.md` — the client's decisions, phrased in the language of obligations:

```
---
id: ACTZ-001
title: "TZ Clarification Protocol No. 1 for TZ-2026-001"
type: ACTZ
tz-ref: TZ-2026-001
resolves: [B-001, B-002, B-003]
status: signed
client-signature: { signed-by: "J. Ivanov", role: PM, organization: "ClientCo",
signed-at: "2026-05-18T15:30:00Z" }
vendor-signature: { signed-by: "P. Petrov", role: architect, organization:
"VendorCorp", signed-at: "2026-05-18T15:40:00Z" }
---

## §1. Sign-in before email confirmation
The system shows a "confirm your email" message and a resend button. A refusal
with no explanation is not allowed.

## §2. Data storage
Data is stored in a data center on RF territory; the jurisdiction is the RF. The
choice of provider is left to the contractor.
```

```
## §3. Resending the email
No more than once every 5 minutes and no more than 5 times per day.
```

The signed protocol comes back into the ADAPT: each finding receives a `decided-in` — a pointer to a clause of a **signed** ACTZ (§7.4.5). Without such a pointer a finding cannot become `resolved`.

```
### B-001: resolved (decided-in: ACTZ-001 §1)
→ a scenario added to Forward §2; SR-04 created.

### B-002: resolved (decided-in: ACTZ-001 §2)
→ data-residency: ["RU"] in Forward §2.

### B-003: resolved (decided-in: ACTZ-001 §3)
→ rate-limit rules in Forward §2; SR-04 refined.
```

All backward entries → `resolved`, `open-questions-count = 0`. **QG-ADAPT-approve** (≡ QG-3) — all preconditions of §10.8.3 are met. The ADAPT is an internal interpretation document, so it is signed by the **Architect alone**; the client's signature lives on the ACTZ (§7.5):

```
status: approved
approval:
  # no client-signature: an ADAPT is an interpretation, not an obligation
  architect-signature: { signed-by: "P. Petrov", role: architect, signed-at:
"2026-05-18T16:00:00Z" }
ai-provenance: { human-edits: true }    # the architect edited the AI draft
open-questions-count: 0
resolved-questions-count: 3
```

After approval the ADAPT is **immutable** (frozen). All BR/SR/SPEC reference the approved ADAPT, not the TZ directly.

The **effective TZ** = the initial TZ + all signed ACTZs (§7.14). It is against the effective TZ — not against the ADAPT — that the system is later delivered and accepted.

Step 4. BR-01 (3 min)

`br/BR-01-user-registration.md`:

```
---
id: BR-01
title: "User registration via email"
type: BR
status: approved
priority: must
source: { adapt: "ADAPT-001", adapt-section: "Forward §2", tz-section: "§2" }
```

```
business-context:
  stakeholder: "J. Ivanov (PM, ClientCo)"
  business-goal: "Enable users to create an account and gain access to the
product"
business-outcome:
  measurement-type: kpi
  kpi-name: "registration-conversion-rate"
  measurement-method: "registered / visited_signup * 100%"
  baseline-value: 0
  target-value: 60
  target-met-by: "2026-09-01"
data-classification: { contains-pii: true, retention-days: 2555, data-residency:
["RU"] }   # 7 years per Ф3-152
compliance: [{ standard: "Ф3-152", article: "ст.6,12" }]
ai-provenance: { generated-by: "anthropic-claude-opus-4-7@2026-05-18", human-
edits: true }
---

# BR-01: User registration via email

A user MUST be able to create an account on their own via email/password
and gain access to the personal dashboard after email confirmation.
```

Step 5. SR-01..SR-04 (3 min)

Decompose the BR into verifiable SRs. Each SR references the approved ADAPT.

`sr/SR-01-sign-up.md` (frontmatter + body):

```
---
id: SR-01
title: "Sign-up via email/password"
type: SR
status: approved
parent: { id: BR-01 }
source: { adapt: "ADAPT-001", adapt-section: "Forward §2", tz-section: "§2" }
constrained-by: ["SPEC-API-01", "SPEC-DATA-01", "SPEC-SEC-01"]
quality-characteristic: [functional-suitability, security]
---

## Description
POST /auth/sign-up accepts {email, password}.
- Valid data → User status `unverified`, a verification email is sent, 201.
- Invalid email (regex/format) → 422 with the field indicated.
- Email already taken → 409.
- Weak password (< 8 chars or in the blacklist) → 422.
```

Likewise — SR-02 (email verification), SR-03 (sign-in for verified users), SR-04 (email resend with the rate limit from B-003).

Step 6. SPEC-* (3 min)

`specs/api/SPEC-API-01-auth.md` (excerpt):

```
---
id: SPEC-API-01
title: "Authentication REST API"
type: SPEC-API
status: approved
source: { adapt: "ADAPT-001", adapt-section: "Forward §2", tz-section: "§2" }
api-style: rest
api-version: "v1.0.0"
versioning-strategy: url-path
authentication: bearer-jwt
rate-limits: [{ endpoint: "POST /auth/resend-email", limit: "1/5min/user;
5/24h/user" }]
contract-file: { format: openapi-3.1, location: "contracts/auth-api.yaml" }
depends-on: ["SPEC-DATA-01", "SPEC-SEC-01"]
referenced-by: ["SR-01", "SR-02", "SR-03", "SR-04"]   # auto-derived
---
```

Likewise — SPEC-DATA-01 (the User schema), SPEC-SEC-01 (the auth model + Φ3-152 controls).

Step 7. TC pos/neg pairing (5 min)

Each SR — at least 1 positive + 1 negative TC. The canonical schema — reference/02 §8.

`tests/TC-01-signup-success.md` (positive, for SR-01):

```
---
id: TC-01
title: "Sign-up: successful registration"
type: TC
tc-type: system
status: ready
verifies: [{ id: SR-01, requirement-version: "1.0" }]
negative: false
automation: { status: automated, location:
"tests/auth/test_signup.py::test_signup_success", runner: pytest }
---

## Given
- a new email (uniquely-generated@test.com); a valid password (length ≥ 8, not in
the blacklist).

## When
POST /auth/sign-up {email, password}

## Then
```

```
- status 201; body {"user_id": "<uuid>", "status": "unverified"}; a User in the
DB with status `unverified`; a verification email sent (mock).
```

`tests/TC-02-signup-invalid-email.md` (negative, for SR-01):

```
---
id: TC-02
title: "Sign-up: reject an invalid email"
type: TC
tc-type: system
status: ready
verifies: [{ id: SR-01, requirement-version: "1.0" }]
negative: true
automation: { status: automated, location:
"tests/auth/test_signup.py::test_signup_invalid_email", runner: pytest }
---

## Given
- email "not-an-email" (no `@`); any valid password.

## When
POST /auth/sign-up {email, password}

## Then
- status 422; body {"field": "email", "error": "invalid format"}; no User created
in the DB; no email sent.
```

Likewise, pairs for SR-02, SR-03, SR-04. For SR-04 (with the rate limit) — an additional TC that checks the block after the 5th attempt within 24 hours.

Step 8. Run the TCs and promote the SR (2 min)

The test runner on the substrate runs the TCs and updates `last-run`:

```
# tests/TC-01-signup-success.md (after the run):
last-run: { date: "2026-05-19T10:00:00Z", result: pass, runner-id: "test-
runner@1.0", run-ref: "<link>", requirement-version: "1.0" }
```

When **all** TCs from `SR-01.verified-by[]` are green and each has a recorded red → green transition (standard/09 §9.18.2) → the engineer's spot-check (§9.14): a sample of 5 passing TCs, directed by machine signals (TCs without red history first), checked against the SR. If the spot-check passes → **QG-2 Verification Gate** passed → SR-01 → `verified`:

```
# sr/SR-01-sign-up.md:
status: verified
```

```
verified-by: ["TC-01", "TC-02"]
verified-at: "2026-05-19T14:00:00Z"
verified-by-engineer: "P. Petrov"
```

Likewise — SR-02, SR-03, SR-04. When all SRs in BR-01 are verified → the BR is ready to be submitted.

Conformance to the **contract** is checked by a separate layer — **AT** (standard/09 §9.19): acceptance tests that an isolated agent derives from the effective TZ alone, without looking into the ADAPT, the SRs, or the code. TCs prove that the system matches the interpretation; ATs prove that the interpretation matches the contract. Until every AT is green, the product is not submitted for delivery (§10.4.3).

Traceability chain

At any moment the provenance of an artifact can be reconstructed:

```
TC-02 (negative)
  └─ verifies SR-01 (sign-up)
       └─ derived from ADAPT-001 §2 Forward (email/password)
            └─ interprets TZ-2026-001 §2 (immutable)
            └─ B-001 decided-in ACTZ-001 §1 (signed: client + contractor)
       └─ constrained-by SPEC-API-01 (REST contract)
                                 └─ depends-on SPEC-DATA-01, SPEC-SEC-01

AT-01
  └─ verifies TZ-2026-001 §2 + ACTZ-001 §1   # the contract only, no internal
artifacts
```

An audit a year later: "where did the requirement to return 422 on an invalid email come from" — TC-02 → SR-01 → ADAPT-001 §2. The trace is complete.

What you just did

- Created an immutable contractual artifact (the TZ).
- Went through two-way interpretation via ADAPT with three backward findings → put the questions to the client as an ACTZ protocol → the signed decisions came back into the ADAPT via `decided-in` → approved.
- Obtained the effective TZ (the initial TZ + the signed ACTZ-001) — the benchmark for the future delivery and acceptance.
- Decomposed it into a BR + 4 SRs, bound to the approved ADAPT.
- Described 3 SPECs (API, DATA, SEC) as the parallel structural axis.
- Covered each SR with a pos+neg TC pair (at least 8 TCs).
- Passed two quality gates: QG-ADAPT-approve (≡ QG-3 Architecture) and QG-2 Verification Gate.

This is the full RENAR cycle in minimal form. On a real project — more BR/SR/SPEC, more backward findings, delegation to AI agents; optionally QG-4 (acceptance).

In real work, steps 2-7 are performed by an AI agent. The engineer does not write the frontmatter and body of artifacts line by line — they frame the task, read the result, refine it, and approve. This is the

regular mode (standard/00 §0.2.1). The full scenario of an initial TZ and a delta-TZ with an adversarial reviewer — in 01-walkthrough.md phase 2 + phase 8.

What next

You want to...	Document
A detailed end-to-end example (login + 2FA + 9 phases)	01-walkthrough.md
Transition from a legacy approach to RENAR	02-transition-guide.md
Git as a substrate (commit policy, PR review, hooks)	03-tool-guide-git.md
A document-oriented substrate	04-document-store-substrate.md
A comparison of RENAR with SAFe / BABOK / ISO 29148	05-safe-comparison.md
Compliance: GDPR / Ф3-152 / AI Act	06-compliance.md
Failure modes — typical failures and patterns	07-failure-modes.md
The full normative specification (15 chapters)	`standard/`
Artifact schemas and validation rules	`reference/02-schemas.md`
Glossary and mapping to industry standards	`reference/01-glossary.md`

RENAR Quickstart 1.0 — renar.tech

01. Walkthrough: Login Flow for AcmeCorp

> *One full RENAR cycle from a signed TZ to an accepted release. The example is an internal tool with registration via corporate email and 2FA. The goal is to show **every phase** on a single medium-sized project.*
>
> ***Context:** AcmeCorp, ~1 sprint of team work, stack Next.js + FastAPI + PostgreSQL. RENAR maturity at the RENAR-3+ level (full ADAPT + TC + adversarial). The example is **substrate-independent**: operations go through the V1–V6 capabilities; the concrete directory layout is in 03-tool-guide-git or 04-document-store-substrate.*
>
> ***Prerequisites:** 00-quickstart, core/renar-core, reference/01-glossary.*

Reader's route. Phases 0–2 — context gathering, signing the TZ, the ADAPT and the TZ clarification protocol (ACTZ). Phases 3–4 — decomposition into BR/SR/SPEC and generation of pos/neg pairs for TC. Phase 5 — the canonical gates QG-0 (requirement approval) and QG-1 (the TC `draft → ready` transition only). Phases 6–7 — TR implementation and verification (QG-2). Phases 8–9 — a delta-TZ on changes and acceptance: AT against the effective TZ plus the optional QG-4 on the business outcome.

Phase 0 — Requirements elicitation

Before signing the TZ. The AI agent runs 2–3 interviews with stakeholders (Sales Director, IT Manager) and gathers context in structured form.

Phase 0 artifacts (informative, not normative for RENAR Core): `elicitation/{domain-context.md, sales-director.yaml, it-manager.yaml, findings-clustered.md, critic-review.md, multi-model-diff.md}`. Phase 0 is not fixed in Core — it belongs to the elicitation methodology, out of scope for RENAR v1.0 (standard/01 §1.3).

Phase 1 — TZ import

After the elicitation iterations, the client signs `TZ-2026-042`:

```
# TZ-2026-042 — Login Flow for AcmeCorp Internal Tool
Signing date: 2026-05-03 · Parties: AcmeCorp + VendorCorp

## §1. Goals
Cut AcmeCorp employees' time to enter the tool to <2 minutes from first arrival
to full access.

## §2. Functional requirements
### FR-001. Registration by corporate email
An employee registers via an email in the @acmecorp.com domain. An email outside
the domain — denial with an explanation.

### FR-002. Two-factor authentication (TOTP)
```

```
After registration, mandatory 2FA setup via TOTP.

### FR-003. Access recovery via the corporate administrator
On loss of the 2FA device — recovery via a ticket in IT support.

## §3. Non-functional requirements
### NFR-001. Performance: Login <2 seconds (p95).
### NFR-002. Security: bcrypt cost-factor ≥ 12; login logs — 1 year; lockout
after 5 failed attempts in 15 minutes.
### NFR-003. Jurisdiction: all data in the RU (government contracts).
```

The TZ is signed → **immutable**. Any edits go through ADAPT (Phase 2) or a delta-TZ (Phase 8). The runtime MUST register the immutable TZ as a revision (V1+V2) with AI provenance (V6).

Phase 2 — ADAPT and ACTZ (interpretation and the clarification protocol)

This is where the two loops part ways. The **ADAPT** is the internal interpretation: its audience is the engineer, the agent, and the reviewer, and it is signed by the Architect (standard/07 §7.5). The **ACTZ** is the TZ clarification protocol: whatever was put to the client and signed by both parties — that is, an obligation (§7.13). The boundary runs along the audience: what is shown to the client and approved is an obligation; what is not shown is an interpretation.

2.1 The primary agent generates a draft ADAPT. Input: TZ-2026-042 (immutable). Output: draft ADAPT-001 + Forward sections (across §2 FR + §3 NFR) + Backward findings (6 candidates) + V6 provenance.

2.2 Adversarial review. A separate critic agent (a different model) checks the backward findings; it blocks adapt-approve while critical findings remain open. Examples:

```
[HIGH] B-001 reclassify gap → hidden-assumption
[HIGH] missed backward: case-sensitivity email in FR-001
[MEDIUM] B-004 terminology: define "employee" via User.role
[MEDIUM] B-006 feasibility: rate-limit scope (IP vs email vs session)
```

2.3 Iterative resolution. The Architect adjusts the Forward and Backward, the AI regenerates. After 2 adversarial cycles: 7 backward entries (B-001..B-007), all reclassified or prepared to be put to the client; the Forward covers §2 + §3 of the TZ in full.

2.4 ACTZ-001 — TZ Clarification Protocol No. 1. Five findings out of seven require a decision from the client. The Architect does not forward the agent's raw output: they aggregate the questions, rephrase them in the language of obligations ("the domain is compared case-insensitively"), attach proposals — and put them forward as a single protocol. The client signs; so does the contractor.

```
---
id: ACTZ-001
title: "TZ Clarification Protocol No. 1 for TZ-2026-042"
type: ACTZ
tz-ref: TZ-2026-042
resolves: [B-001, B-002, B-004, B-006, B-007]
```

```
status: signed
client-signature: { signed-by: "A. A. Ivanova", role: "Product Lead",
organization: "AcmeCorp", signed-at: "2026-05-04T11:30:00Z" }
vendor-signature: { signed-by: "P. P. Petrov", role: architect, organization:
"VendorCorp", signed-at: "2026-05-04T11:50:00Z" }
---

## §1. Email case
`@AcmeCorp.com` and `@acmecorp.com` denote the same employee: the domain is
compared case-insensitively.

## §2. The term "employee"
An employee is an account with the role `employee` in the corporate directory.

## §3. The lockout boundary
5 failed attempts within 15 minutes are counted per "IP + email" pair, not per
session.
```

Every finding that requires a decision from the client receives a `decided-in: ACTZ-001 §M` — a pointer to a clause of the **signed** protocol. A finding cannot become `resolved` without such a pointer, and an ADAPT cannot be approved while findings remain unclosed (§7.4.5).

2.5 ADAPT in status `approved`. The signature is the Architect's alone: the client never saw this document and never signed it — their decisions live in ACTZ-001.

```
---
id: ADAPT-001
title: "Adaptation of TZ-2026-042 — Login Flow AcmeCorp"
type: ADAPT
trigger-stage: import-tz
source-tz: { id: TZ-2026-042, signed-date: "2026-05-03", signed-by-client:
"AcmeCorp PM" }
status: approved
approval:
  # no client-signature per §7.5 — the ADAPT is not put to the client
  architect-signature: { signed-by: "P. P. Petrov", role: architect, signed-at:
"2026-05-04T12:00:00Z" }
generates-requirements: [BR-01, BR-02, SR-01, SR-02, SR-03, SR-04, SR-05, SR-06,
SR-07]
generates-specs: [SPEC-UI-01, SPEC-API-01, SPEC-DATA-01, SPEC-SEC-01]
open-questions-count: 0
resolved-questions-count: 7
ai-provenance: { generated-by: "anthropic-claude-opus-4-7@2026-05-04", prompt-
template: "prompts/adapt-from-tz.md@v2.1", human-edits: true }
---
```

Once approved, ADAPT-001 is **immutable**.

The **effective TZ** (§7.14) at this point: `TZ-2026-042` + `ACTZ-001`. That is the benchmark the system will be delivered against in phase 9. The ADAPT is not part of the acceptance benchmark — the client answers only for what they signed.

Phase 3 — Decomposition into BR / SR / SPEC

3.1 Decomposition. Operation: `decompose`. Input: approved ADAPT-001. Output: draft BR (2), SR (7), SPEC (4) + adversarial-review artifacts.

3.2 Adversarial findings:

```
[HIGH] BR-01 stakeholder field empty — who owns the business goal?
[HIGH] SR-05 says "bcrypt cost-factor 12" — this is a deployment detail, it
belongs in SPEC-SEC-01, not in the SR.
[MEDIUM] NFR-003 (jurisdiction) is not reflected in the data-classification of
SR-01.
[MEDIUM] SPEC-UI-01 has no accessibility-level — WCAG-AA minimum for a corporate
tool.
→ 4 findings → fix → re-generate.
```

3.3 Final artifact set:

```
acmecorp-requirements/
├── br/
│   ├── BR-01-self-service-registration.md
│   └── BR-02-secure-mfa.md
├── sr/
│   ├── SR-01-email-domain-validation.md       (FR-001)
│   ├── SR-02-totp-enrollment.md               (FR-002 setup)
│   ├── SR-03-totp-verification.md             (FR-002 verify)
│   ├── SR-04-password-recovery-via-admin.md   (FR-003)
│   ├── SR-05-rate-limiting-failed-logins.md   (NFR-002)
│   ├── SR-06-audit-logging.md                 (NFR-002 audit)
│   └── SR-07-data-residency-ru.md             (NFR-003)
├── specs/
│   ├── ui/SPEC-UI-01-login-flow.md
│   ├── api/SPEC-API-01-auth.md
│   ├── data/SPEC-DATA-01-user-model.md
│   └── sec/SPEC-SEC-01-auth-policy.md
└── tz/TZ-2026-042.md
```

3.4 Example: SR-01 (frontmatter + body):

```yaml
---
id: SR-01
title: "Email domain validation at registration"
type: SR
status: approved
parent: { id: BR-01 }
source: { adapt: "ADAPT-001", adapt-section: "Forward §2.1", tz-section: "§2 FR-
001" }
```

```
constrained-by: ["SPEC-API-01", "SPEC-DATA-01", "SPEC-SEC-01"]
data-classification: { contains-pii: true, data-residency: ["RU"], retention-
days: 365 }
compliance: [{ standard: "FZ-152", article: "art. 13.1" }]
ai-provenance: { generated-by: "anthropic-claude-opus-4-7@2026-05-04", prompt-
template: "prompts/decompose-adapt.md@v2.1", context-tokens: 12450, output-
tokens: 320, human-edits: true }
---

## Description
Registration is allowed only if the email belongs to the `@acmecorp.com` domain.
Other domains are rejected with an explanation.

## Behavior
- email NOT from `@acmecorp.com` → 422 with `{"error":"email-domain-not-allowed",
"allowed-domain":"acmecorp.com"}`. [ADAPT-001 Forward §2.1; TZ-2026-042 §2 FR-
001]
- email from `@acmecorp.com` → standard registration (SPEC-API-01).
- The whitelist is stored in `SPEC-SEC-01.allowed-domains`, extended without a
release.
- Domain comparison is case-insensitive (ADAPT-001 Forward §2.1).

## Constraints
- `*.acmecorp.com` subdomain — a separate Architect decision (out of scope).
```

3.5 SPEC-API-01 (fragment):

```
---
id: SPEC-API-01
title: "Authentication REST API"
type: SPEC-API
status: approved
source: { adapt: "ADAPT-001", adapt-section: "Forward §2" }
api-style: rest
api-version: "v1.0.0"
versioning-strategy: url-path
authentication: bearer-jwt
rate-limits: [{ endpoint: "POST /auth/login", limit: "5/15min/ip+email" }]
contract-file: { format: openapi-3.1, location: "contracts/auth-api.yaml" }
depends-on: ["SPEC-DATA-01", "SPEC-SEC-01"]
---

## Endpoints

### POST /auth/register
- body: `{"email": "<corp-email>", "password": "<strong>"}`
- 201 → `{"user_id": "<uuid>", "verified": false, "totp_setup": false}` · 422 →
invalid · 409 → email exists

### POST /auth/login
- body: `{"email", "password", "totp": "<6digits>"}`
- 200 → `{"access_token": "<jwt>", "expires_in": 3600}` · 401 → invalid · 429 →
```

```
rate limit

### POST /auth/totp-setup — see SR-02.

## Error model
A single structure: `{"error": "<code>", "details": {...}}`.
```

Phase 4 — TC generation (pos/neg pairs)

Operation: `tc-generate` SR-01 → pos/neg TC pairs per testable assertion (standard/09 §9.7: for each assertion in an SR — at least one positive + negative TC pair).

TC-001 (positive):

```
---
id: TC-001
title: "Registration with an allowed domain — happy path"
type: TC
tc-type: system
status: ready
verifies: [{ id: SR-01, requirement-version: "1.0" }]
negative: false
automation: { status: automated, location:
"tests/auth/test_registration.py::test_allowed_domain_succeeds", runner: pytest }
---

## Given
- The DB is empty; email alice@acmecorp.com is not registered.

## When
POST /auth/register {email: "alice@acmecorp.com", password: "ValidPass123!"}

## Then (Pass)
- status 201; body contains {"user_id": "<uuid>", "verified": false,
"totp_setup": false}; the User is created in the DB; a verification email is sent
(mock SES).

## Fail criteria
- status ≠ 201; plaintext password in the body/logs; a User with a different
email (case mismatch); the verification email is not sent.

## Not in scope
- TOTP setup → TC-005 (SR-02); rate limiting → TC-009 (SR-05).
```

TC-004 (negative):

```
---
id: TC-004
title: "Registration with a disallowed domain — denial with an explanation"
```

```
type: TC
tc-type: system
status: ready
verifies: [{ id: SR-01, requirement-version: "1.0" }]
negative: true
automation: { status: automated, location:
"tests/auth/test_registration.py::test_disallowed_domain_rejected", runner:
pytest }
---

## Given
- email "bob@gmail.com" (outside the whitelist).

## When
POST /auth/register {email: "bob@gmail.com", password: "ValidPass123!"}

## Then (Pass)
- status 422; body == {"error": "email-domain-not-allowed", "allowed-domain":
"acmecorp.com"}; the User is NOT created in the DB; no email is sent; an audit
entry about the rejected attempt (for SR-06).

## Fail criteria
- status ≠ 422; the User is created; an email is sent (security leak); the audit
entry is missing.
```

Phase 5 — Quality gates before code (QG-0 and QG-1)

After generating TCs for all 7 SRs — 26 test-case entries in total (pos/neg pairs + extra negatives for SR-05, SR-06).

5.1 QG-0 — approval of BR-01 (`draft → approved`). Preconditions: `source.adapt = ADAPT-001 (approved)`; the SR tree in an admissible state before the BR is approved; the adversarial review succeeded; the assertions and links cite sections of ADAPT-001. Postcondition: BR-01 + child SRs, where needed, cascade `draft → approved`.

5.2 QG-1 — TC only: `draft → ready`. Per §10.3.2, QG-1 applies only to TC and separates a prepared test case with an executable verification implementation from a draft. Preconditions for a TC: the implementation version-pin is fixed (V5); `automation.status` + `location` are valid; static checks pass; pos/neg pairing (§9.7); the mandatory body sections are filled in. Postcondition: `draft → ready`.

After **QG-0** on BR/SR and **QG-1** on each TC, TR work can be opened (Phase 6).

Phase 6 — Implementation

> ***The order is not accidental.*** *The TCs were frozen (`ready`) in phase 5 — **before** the first line of code, and they were not written by the agent that now writes the implementation (standard/09 §9.18). A test born next to the code checks the code, not the requirement. Hence the red-history requirement as*

> *well: before implementation the TC is run and MUST fail — a test that has never been red proves nothing.*

6.1 Task creation (TR). Operation `sync-tasks` : input — the verified SR/SPEC set; output — 7 TRs in the implementation tracker (parent SR, implements-spec[], QG-0 ready with Goal + AC).

6.2 A developer picks up TR-101. QG-0 checks: Goal from SR-01; AC list (4 items); parent.id resolves (approved); implements-spec present; negative scenario in AC → the work session is allowed.

6.3 Implementation (fragment):

```
# acmecorp-login.src/src/auth/registration.py
from fastapi import HTTPException
from config import settings   # allowed-domains from SPEC-SEC-01

def validate_email_domain(email: str) -> None:
    domain = email.split("@", 1)[-1].lower()
    if domain not in settings.AUTH_ALLOWED_DOMAINS:
        raise HTTPException(status_code=422, detail={
            "error": "email-domain-not-allowed",
            "allowed-domain": settings.AUTH_ALLOWED_DOMAINS[0],
        })
```

```
# acmecorp-login.src/tests/auth/test_registration.py
def test_allowed_domain_succeeds(client, db, mock_ses):
    r = client.post("/auth/register", json={"email": "alice@acmecorp.com",
"password": "ValidPass123!"})
    assert r.status_code == 201
    assert "user_id" in r.json()
    user = db.query(User).filter_by(email="alice@acmecorp.com").one()
    assert user.verified is False
    mock_ses.send_email.assert_called_once_with(template_id="verification-email",
to_email="alice@acmecorp.com")

def test_disallowed_domain_rejected(client, db):
    r = client.post("/auth/register", json={"email": "bob@gmail.com", "password":
"ValidPass123!"})
    assert r.status_code == 422
    assert r.json() == {"error": "email-domain-not-allowed", "allowed-domain":
"acmecorp.com"}
    assert db.query(User).count() == 0
```

6.4 Substrate-side validation hook:

```
[hook] Checking links for TR-101: parent.id SR-01 (approved); implements-spec
[SPEC-API-01, SPEC-SEC-01].
[hook] Negative TCs: SR-01.verified-by includes TC-002, TC-004 (negative).
✓ Change allowed.
```

Phase 7 — QG-2 (verification gate)

7.1 CI runs the TCs. `pytest acmecorp-login.src/tests/auth/test_registration.py` → 4 TC PASSED → the Bot updates `last-run.result = pass`, `requirement-version = 1.0` in the TC files. Each TC's run history shows the "red → green" transition: the pinning run before implementation was red — the condition under which a TC counts as evidence at QG-2 (§9.18.2).

7.2 Spot-check (standard/09 §9.14). Once per sprint the Engineer manually runs 5 passing TCs selected by machine signals (priority — TCs with no red history, then mutant survivors, then the `implementation-originated` class; the remainder by random top-up), and checks the actual result against the SR. Selected: TC-001, TC-008, TC-012, TC-019, TC-024 → 5/5 match.

7.3 Promote SR-01 → verified. QG-2 preconditions: approved ADAPT linkage; pos/neg TCs passing; `last-run.requirement-version` is fixed; the spot-check passed. Postcondition: SR-01 `approved → verified`; the coverage index is updated.

Phase 8 — Delta-TZ

8.1 The client, a week later:

```
# TZ-2026-051 — Addendum to TZ-2026-042
Base: TZ-2026-042

## §2 (change) FR-001 (extension)
Additionally allow @subsidiary.acmecorp.com (the subsidiary company). The
whitelist is extended to 2 domains.
```

8.2 Delta-ADAPT + ACTZ-002. Operation `adapt-from-tz (delta)`: input — TZ-2026-051 + parent ADAPT-001; output — draft ADAPT-001-delta-1 + delta Forward + backward findings (e.g. B-008 scope). Finding B-008 ("is `@sub.subsidiary.acmecorp.com` to be treated as allowed?") is put to the client as the protocol `ACTZ-002` — "TZ Clarification Protocol No. 2"; once signed, B-008 receives `decided-in: ACTZ-002 §1`, and the delta-ADAPT is approved by the Architect's signature. The effective TZ becomes: `TZ-2026-042` + `TZ-2026-051` + `ACTZ-001` + `ACTZ-002`.

8.3 Impact analysis. Operation `impact-analysis --delta TZ-2026-051`:

```
Affected:
  BR-01 (scope extension)
  SR-01: verified → approved (TC rerun required)
  TC-001..004: re-pin 1.0 → 1.1; +2 new TC (subsidiary domain)
  TR-115: new implementation task
  SPEC-SEC-01: allowed-domains extended
```

8.4 Apply delta. The Architect opens the changes with the marker `[delta:TZ-2026-051]`. The AI updates SR-01 (extends the whitelist) and generates 2 new TCs. Implementation in TR-115. CI runs the TCs, the bot updates last-run. After the spot-check — SR-01 v1.1 → verified again.

> ***Note (simple delta).*** *If the adversarial reviewer returns a "no findings, no clarifications" verdict (§7.4.1.2),* ***no delta-ADAPT is created*** *— BR/SR/SPEC get* `source.tz-section` *directly with a fixed* `adversarial-review-ref`*. A trivial change (a field rename) produces neither an interpretation nor a protocol: there is nothing to ask the client about.*

Phase 9 — Acceptance: AT against the effective TZ + QG-4

9.1 Regenerating the ATs. Before the trials, an isolated agent re-derives the acceptance tests — **AT** — from the current revision of the effective TZ (standard/09 §9.19). Its input is only `TZ-2026-042` + `TZ-2026-051` + `ACTZ-001` + `ACTZ-002`. It has no access to the ADAPT, the BR/SR/SPEC, the TCs, or the code, and its model differs from the primary agent's — otherwise it would reproduce the same interpretation and acceptance would degenerate into a second round of verification.

```
---
id: AT-03
title: "Registration from the subsidiary domain"
type: AT
verifies: [{ tz: "TZ-2026-051 §2" }, { actz: "ACTZ-002 §1" }]
tz-version: "effective@2026-05-14"
generator: { vendor: "<vendor-B>", model: "<model-B>", internal-context-access:
false }
negative: true
status: passing
environment-ref: SPEC-TEST-01
automation: { kind: dynamic, location: "at/test_subsidiary_domain.py", runner:
pytest }
---

## tz_text
"Additionally allow @subsidiary.acmecorp.com (a subsidiary). The whitelist is
extended to 2 domains."
```

A verbatim quote of the contract clause next to the verification steps is a mandatory part of the AT body: at the trials it is the clause of the contract itself that is presented, not a paraphrase of it.

9.2 The release gate. The product is not submitted for delivery until every AT is in status `passing` (§10.4.3). The run gives 11/12 green: `AT-07` ("access recovery via the administrator") is red while every TC is green. That is the most valuable row of the routing table (§9.19.5): AT ✗ / TC ✓ — an **interpretation** error, not a code error. The analysis confirms it: the ADAPT reduced recovery to a password reset, whereas the TZ requires a ticket to IT support. The fix goes into the ADAPT (errata) and the derived SR/TC, after which AT-07 turns green. The `ACCEPTANCE.md` report — AT coverage across the sections of the effective TZ — is refreshed automatically.

9.3 QG-4 (optional) — the business outcome. Four weeks after release the business effect is measured. The gate does not replace contractual acceptance and is not mixed with it: the goal can be met while the contract is not, and vice versa.

```
BR-01 KPI: Time-to-first-login — target <2min P95, actual 1.4min (143%)
BR-02 KPI: 2FA adoption — target ≥95%, actual 97%
```

9.4 Sign-off. The client accepts the result. BR → status `accepted`. Archive: `ACCEPTANCE.md` + `QG-4-REPORT-v1.0.md` + lessons learned `lessons/2026-Q2.md`.

Final artifacts

```
acmecorp-requirements/
├── adapt/                  ADAPT-001-main.md (frozen) + ADAPT-001-delta-1.md
(frozen)
├── actz/                   ACTZ-001.md + ACTZ-002.md (signed, immutable)
├── br/                     BR-01 + BR-02 (status: accepted)
├── sr/                     SR-01 (v1.1, verified) + SR-02..SR-07 (v1.0, verified)
├── specs/                  SPEC-UI-01 + SPEC-API-01 + SPEC-DATA-01 + SPEC-SEC-01
(verified; SPEC-SEC-01 v1.1)
├── tests/                  TC-001..TC-028 (28 TC, 100% passing)
├── at/                     AT-01..AT-12 (regenerated before the trials, 100%
passing)
├── tz/                     TZ-2026-042.md + TZ-2026-051.md (delta, immutable)
├── elicitation/            # Phase 0 artifacts
├── lessons/2026-Q2.md      # Phase 9 lessons
├── ACCEPTANCE.md           # AT coverage across the effective TZ
└── QG-4-REPORT-v1.0.md     # business-outcome report
```

The effective TZ here is not a file but a computed sum: `TZ-2026-042` + `TZ-2026-051` + `ACTZ-001` + `ACTZ-002`. That is exactly what was fed to the AT generator.

Project metrics

Metric	Value
RDLT (TZ signed → all SR verified)	11 days
Coverage Velocity	100% over 2 sprints
Hallucination Rate (detected)	0%
Adversarial findings found (cycle 1)	4 high + 2 medium
Test-spec drift on the delta-TZ	0%

Signed ACTZs	2 (TZ Clarification Protocols No. 1 and No. 2)
ATs at the trials	12 (1 red → an interpretation error, fixed)
Acceptance disputes	0 (1 finding, resolved before sign-off)
Cost per BR	$0.46 (gen) + $0.18 (critic) = $0.64
Total AI cost	~$8.50
BRs accepted	2/2
Days to accept	35

What this example shows

1. **Transparency** — every artifact has provenance, every transition is a gate with explicit conditions.
2. **Speed** — decomposing an approved ADAPT — tens of seconds + 2 adversarial cycles.
3. **Traceability** — from a line in the TZ to a passing TC in a handful of substrate query operations.
4. **Delta-TZ** — the affected SR/SPEC/TC/TR are computed automatically.
5. **Two loops instead of one** — the client signs the ACTZ (obligations), the Architect signs the ADAPT (interpretation). The argument "is this a clarification or already a change?" never arises: it is enough to ask whether the decision was put to the client.
6. **Acceptance against the contract** — the ATs, derived by an isolated agent from the effective TZ, caught an interpretation error that all 28 green TCs missed by construction.
7. **Closing the loop** — QG-4 ties the result to business metrics (KPI achievement).
8. **AI-nativeness** — the critic and the generator are different models (isolation); the spot-check finds discrepancies that an automated run alone may miss.

What's next

- 02-transition-guide.md — transitioning from a legacy approach.
- 03-tool-guide-git.md — git as a substrate.
- 04-document-store-substrate.md — a document-oriented substrate.
- 05-safe-comparison.md — comparison with SAFe / BABOK / ISO 29148.
- 06-compliance.md — compliance mapping (GDPR / FZ-152 / AI Act).
- 07-failure-modes.md — failure modes.

RENAR Walkthrough 1.0 — renar.tech

02. Transitioning to RENAR

> *Teams rarely start from a blank page. This chapter is about how to move an existing project from the manual "TZ → code" loop to RENAR gradually, step by step, without halting development. Each level delivers tangible value; a team chooses which* `RENAR-N` *to reach based on real value, not on a race for "full conformance" to claims on paper.*
>
> ***Prerequisites:*** *RENAR Core, standard/11-maturity-model (closed list of levels RENAR-1..RENAR-5), guide/07-failure-modes. Already on early RENAR with legacy types (* `INT-TC` *,* `AIC` *, ...) — see 10-migration-v1.*

1. Assessment: where the team is now

Before migrating, assess the current state. A "pre-RENAR" checklist:

Signal	Pre-RENAR	RENAR-1 minimum
The TZ exists as an artifact	In chat / Google Doc / Notion	Fixed in the substrate as a file
Someone maintains the requirements	Only in a tracker (Jira / Linear)	In the substrate as BR / SR / ADAPT
Where a requirement came from	From memory / we re-ask	Anyone can find the source in the substrate
Requirement changes	Verbally at standup	Through an explicit change-set (delta-TZ)
Tests	In code, no link to requirements	TC as artifacts, or at least in code with an SR-ID mention

If 3+ rows fall in the "Pre-RENAR" column, the team is **below RENAR-1**. This is a normal starting point for most projects.

1.1 Readiness signals

A team is ready to migrate if:

- It hurts that requirements get lost between chats / tickets / documents.
- Disputed acceptances happen regularly — "this isn't what we asked for".
- Onboarding a new engineer takes months to reconstruct context.
- AI is used to generate requirements / code, but there is no systematic verification.

If none of this hurts, RENAR may be premature optimization. See §8 "when it is not needed".

2. Stage 1 — entering RENAR-1 (Ad-hoc)

Level goal: requirements live in the substrate, not in chat; every TZ has gone through adversarial review with an issued AR, and an ADAPT is created on a "findings present" verdict.

Typical duration: 1-2 weeks.

2.1 What gets added

1. An artifact **substrate** is chosen and set up: a `<project>.req/` directory in the repository, a document-store workspace, or another substrate — RENAR is not tied to a specific implementation; see capabilities **V1–V6** (glossary §2.7).
2. The existing TZ is moved into this environment as an artifact not subject to arbitrary edits (with date and signatures).
3. Every TZ goes through adversarial review; its outcome is issued as an `AR` — an adversarial-review record (standard/07 §7.4.6). On a "findings present" verdict an `ADAPT` is created — a bridge artifact: a forward-interpretation section ("how we understood it") and a backward-findings section. On a "no findings" verdict no ADAPT is created, and BR / SR / SPEC reference the TZ directly through `source.tz-section` + `source.adversarial-review-ref` (§7.4.1).
4. New requirements are recorded as BR / SR files in the substrate, not only in the tracker.

2.2 What is NOT required at this stage

- Standardized frontmatter (at minimum — `id` + `title`).
- Lifecycle statuses (`draft` / `approved` /...) — artifacts MAY live without explicit transitions.
- TC as separate artifacts.
- Substrate hooks.
- Substrate-native COVERAGE.

2.3 Common blockers

- **"We already have tickets in Jira"** — leave them. RENAR-1 does not require migration; the tracker and the substrate can coexist. The key point is that the substrate is now the Source of Truth for **new** requirements.
- **"ADAPT is extra work"** — an ADAPT takes 1-2 hours per TZ. It pays off at the very first disputed acceptance.
- **"Where do we store it?"** — any substrate that satisfies **V1–V6**. See guide/03-tool-guide-git or guide/04-document-store-substrate.

2.4 When to move to RENAR-2

When **new requirements** have stopped going into chats and started appearing in the substrate reliably. This takes 4-6 sprints of habit-building.

---

3. Stage 2 — moving to RENAR-2 (Documented)

Level goal: structure and frontmatter; delta-TZ as an explicit change-set.

Typical duration: 2-4 weeks after RENAR-1.

3.1 What gets added

1. Every BR / SR gets frontmatter with mandatory fields (see reference/02-schemas).
2. Folders are structured: `br/`, `sr/`, `adapt/`, `actz/`, `dpia/`, ...
3. Requirement changes — only through a delta-TZ (a new immutable artifact), not through direct editing.
4. The TZ is fixed as immutable (changes only through a delta-TZ).
5. Clarifications obtained from the client without a new TZ are drawn up as an **ACTZ** — a TZ clarification protocol with a dual signature (standard/07 §7.13) — not as a message in a chat.

What this step really changes. Before RENAR the team has one contractual document — the TZ — and everything discussed after it was signed lives in correspondence. RENAR separates two things that used to be conflated:

What	Where	Who signs
A decision put to the client and accepted by them	ACTZ ("TZ Clarification Protocol No. N")	The client **and** the contractor
The engineer's reading of the TZ — terms, elaborated scenarios, findings	ADAPT	The Architect alone (§7.5)

The client does not sign the ADAPT: that used to be a fiction — a signature under an engineering document the client never read. The practical upshot is the **effective TZ** (§7.14): the initial TZ plus every signed ACTZ. It — not the correspondence and not the ADAPT — becomes the benchmark for delivery and acceptance.

3.2 Template: legalizing existing requirements

Old requirements (already in the substrate since RENAR-1) — run a review and stamp frontmatter "as-is" without revising the content. This is **legalization**, not **rewrite**:

```
---
id: BR-12
title: "Employee registration via corporate email"
status: approved          # already running in prod
created-at: "2025-12-01"  # retroactively
priority: must            # retroactive assessment
legacy: true              # marker: the requirement did not go through ADAPT from
scratch
---
```

The `legacy: true` field is optional but RECOMMENDED — it distinguishes "historical" requirements from new ones that went through the full RENAR pipeline from the start.

3.3 Common blockers

- **"frontmatter for 100 requirements is a month"** — yes. Do it in the background, 10-15 requirements / week; new requirements get frontmatter immediately.

- **"Delta-TZ slows us down"** — in the first weeks, yes. After 2-3 iterations it usually becomes faster than "let's just change it".
- **"We don't know what priority to set retroactively"** — leave `priority: should` for all legacy; an explicit `must` is set only when confirmed with a Stakeholder.

3.4 When to move to RENAR-3

When frontmatter is valid for 80%+ of artifacts and the team has gotten used to delta-TZ.

---

4. Stage 3 — moving to RENAR-3 (Tracked)

Level goal: automatic validation + lifecycle enforcement + TC coverage for priority=must.

Typical duration: 4-8 weeks after RENAR-2.

4.1 What gets added

1. Substrate hooks validate frontmatter against the schema on every change. Invalid ones — integration is blocked.
2. Lifecycle statuses are used for real: every artifact is in one of the closed states (`draft` / `approved` / `verified` / `deprecated` / `obsolete`).
3. TC are created for all priority=must BR / SR / SPEC.
4. A substrate-native COVERAGE report is auto-generated on every promote-transition.
5. Reference-validation hook: creating a BR / SR with a link to an ADAPT in a status below `approved` — blocked.
6. The implementation references BR / SR / SPEC with a pinned `verifies[].version` .

4.2 Template: backfilling TC

For all priority=must requirements without a TC:

1. Sort by "frequency of mention in incident reports" — where a TC delivers the most value.
2. Cover 3-5 requirements / sprint, without trying to backfill everything at once.
3. `TC` are created as artifacts in your substrate with a `verified-by` link.

4.3 Common blockers

- **"Old requirements stubbornly resist the schema"** — leave them in a legacy folder; new ones are schema-valid right away. The substrate hook applies only to new / modified artifacts.
- **"Hooks slow down the PR cycle"** — measure: if > 30s — optimize the hooks (parallelization, caching); do not "turn them off".
- **"The team bypasses hooks via `--no-verify` "** — this is an organizational failure pattern; the root cause is that the hooks are too slow / noisy. Fix them, do not forbid.

4.4 When to move to RENAR-4

When COVERAGE for priority=must = 100%, frontmatter is valid everywhere, and the lifecycle actually works.

5. Stage 4 — moving to RENAR-4 (Verified)

Level goal: all approved artifacts have successfully passing test cases (`TC`); the transition under the `QG-2` gate (`Verification Gate`) is enforced by the substrate; positive / negative pairing is observed; an `ai-provenance` block is set for AI-sourced material.

Typical duration: 6-12 weeks after RENAR-3.

5.1 What gets added

1. 100% of approved artifacts have a `verified-by` link to ≥ 1 TC.
2. Pos/neg pairing for every normative clause.
3. The `QG-2` gate (`Verification Gate`) is blocked by substrate-built-in checks: a transition to `verified` only when all `TC` s pass for the current `requirement-version` .
4. All TC are automated or explicitly `manual-pending` with a deadline.
5. For `tc-type: ux` — VLM-judge isolation.
6. For `tc-type: eval` — the judge model ≠ the implementation model.
7. `ai-provenance` for AI-generated artifacts: at minimum `generated-by` + `generated-at` .
8. Source citation — every normative clause has a pointer to a source in the TZ or ADAPT.
9. Reconciliation is run by the substrate at least once a week.
10. Spot-check 5 random passing TC once a sprint.
11. Test-authorship isolation: a TC is frozen (`ready`) before implementation starts, and it is not written by the agent that writes the code (standard/09 §9.18).
12. Red history: the pinning run of a TC before implementation MUST be red. A test that is green before the code exists is a signal to investigate, not a success.

Acceptance against the contract. At this same stage a second layer of checking appears — **AT** (standard/09 §9.19). TCs prove conformance to the interpretation; ATs prove conformance to the effective TZ. They are generated by an isolated agent whose input is **only** the effective TZ (no ADAPT, SR, TC, or code) and whose model differs from the primary agent's. ATs are regenerated before every round of trials; until every AT is green, the product is not submitted for delivery (§10.4.3).

What this buys in practice: a team with green TCs and a red AT is looking at an error of **interpretation** of the TZ — the class of defect no TC catches, because the TCs were derived from the very interpretation the code was.

5.2 Template: phased coverage

Reaching 100% verified-by coverage is not a single PR. The approach:

1. First pos-only TC for all priority=must (the benefit — fast feedback).
2. Then neg-TC pairs (the benefit — catching test-fitting drift).
3. Then `tc-type` extensions (ux / eval / contract / security) as needed.

5.3 Common blockers

- **"Judge-model isolation is expensive"** — true. But it is a structural rate limit on AIR-06 test-fitting drift — it cannot be circumvented without losing verification integrity.
- **"Source citation slows authors down"** — automate it: the AI generator should emit the citation right away; manual authoring is only for legacy backfill.
- **"Reconciliation findings overwhelm the team"** — tunable thresholds; start with conservative defaults and loosen gradually.

5.4 When to move to RENAR-5

When RENAR-4 runs stably for 2-3 quarters, the metrics are stable, and the team is not "burning out" from reconciliation noise.

---

6. Stage 5 — moving to RENAR-5 (Optimized)

Level goal: adversarial review as a gate; multi-model agreement for priority=must; cost/latency budgets; knowledge graph as primary search; continuous evaluation for AI-critical components.

Typical duration: 12+ weeks after stable RENAR-4.

6.1 What gets added

1. An adversarial critic — a mandatory gate for `draft → approved`. The critic is a model different from the generation model.
2. Multi-model agreement for priority=must: the artifact is generated by ≥ 2 models; divergences are flagged and MUST be reviewed.
3. Cost / latency budget per artifact; an overrun → automatic decomposition.
4. Knowledge graph as primary search for AI agents.
5. Continuous evaluation for SPEC-AI.
6. Hallucination Rate metric < 1%.
7. Multi-model Disagreement Rate metric is tracked.
8. Feeding template improvements back into the `requirements-library` is standard practice.

6.2 When RENAR-5 is needed

- Regulated industries (fintech, healthcare, high-risk AI systems per the AI Act).
- When the product critically depends on AI-generation quality (generative products).
- When there is a budget for continuous-evaluation infrastructure.

Many projects can stop at RENAR-4 — it is enough for **conformance to a declared RENAR profile**. RENAR-5 is not necessarily "better", but it is almost always **more expensive and stricter**. Choose by need, not by fashion.

---

7. Migrating legacy requirements

Thousands of existing requirements in Jira / Confluence — how do you pull them in?

7.1 Non-migration strategy

If legacy requirements are not being actively changed — **do not migrate**. RENAR applies only to new requirements and actively changed ones. Legacy stays in its original place as read-only "historical context".

7.2 Selective-migration strategy

For legacy that is actively changing:

1. At the moment of the first change — pull it into the substrate as a BR/SR with a `legacy: true` marker and minimal frontmatter.
2. Apply the change as a delta-TZ.
3. After 1-2 iterations the requirement "matures" — it becomes full-RENAR (no legacy marker, with full frontmatter and TC).

7.3 Bulk-import strategy

When you need to migrate > 100 requirements at once (for example, during an organizational reorganization):

1. Scripted export from the tracker → a frontmatter skeleton (id, title, minimum).
2. All imported requirements — `legacy: true` + `status: approved` (as in prod).
3. Backfill TC and full frontmatter — sprint by sprint, without blocking current work.

---

8. When RENAR is NOT needed

RENAR is overhead. Do not apply it to:

- **One-off scripts and prototypes** — they never reach production, or they become production only through a rewrite.
- **Very small teams (1-2 people)** — the overhead of managing a substrate may exceed the benefit.
- **Projects with no AI-generation of requirements / tests** — the main value of RENAR (protection against AI-specific failure modes) is lost.
- **Short-lived experiments (≤ 1 sprint)** — you will not have time to recoup the ADAPT setup.

Minimum payback threshold: a project with ≥ 3 requirement iterations, ≥ 2 developers, using AI somewhere in the pipeline (generation of requirements / code / tests / documentation).

---

9. Migration anti-patterns

What to avoid:

9.1 Big-bang migration

Symptom: The team stops development for 2 sprints to "move to RENAR". After 2 sprints they get burnout and an abandoned migration.

Instead: Hybrid mode. New requirements go straight into RENAR, old ones — as they are touched. Slow, but sustainable.

9.2 Skipping levels

Symptom: "Let's go straight to RENAR-4". The team skips RENAR-2/3, puts in full hooks + ai-provenance + an adversarial critic, but frontmatter is not valid and the lifecycle is chaotic.

Instead: Levels go in order. RENAR-4 without a RENAR-3 base does not work.

9.3 "Perfect frontmatter" paralysis

Symptom: The team spends weeks polishing one piece of frontmatter — "did I pick the right `priority`?" The real work stalls.

Instead: frontmatter is "good enough". Mistakes are fixed in a delta-TZ. The point is that something is in the substrate, not that it is buffed to a shine.

9.4 Partial adoption of the substrate

Symptom: Half the requirements are in the substrate, half are still in Jira. Nobody knows where the Source of Truth is.

Instead: The Source of Truth MUST exist **right away**. If you cannot move everything — move only the new ones and explicitly declare the substrate the owner of "everything new".

9.5 "Tooling first" approach

Symptom: The team spends half a year writing internal tooling around RENAR (its own validator / its own CI / its own UI) without using the standard itself.

Instead: Practice first, on a minimal substrate (even a flat folder of markdown). Tooling — when it starts to hurt by hand.

Adoption cost and benefit (illustrative)

> *This section is **illustrative and non-normative** — it helps estimate orders of magnitude. The concrete numbers depend on the project; calibrate the real model against your own data.*

Adoption cost (what you put in):

- Training the team on RENAR Core — on the order of half a day to a day per engineer.
- Adversarial review of every TZ, and ADAPT discipline where the review produced findings — additional hours on backward findings before coding starts (recouped by not re-opening the TZ later).
- The substrate already exists (git) — no separate licenses needed; tooling automation is optional and introduced gradually.

Benefit (what you get back):

- Reduced TZ-decomposition time with AI acceleration (order of magnitude — standard/12 §12.5.1: 5–10× at RENAR-4, illustrative).
- Fewer disputes at acceptance: a signed ACTZ fixes the obligation before code, and delivery runs against the effective TZ rather than against someone's memory of the email thread.
- Fewer incidents from negative scenarios — thanks to mandatory pos/neg TC pairing.
- An audit log of "what we delivered under contract" — free from the substrate (V1 / V6).

When it does not pay off: short-lived prototypes, pure product discovery without a contract, teams with no compliance pressure and no external client (see §8 "when RENAR is not needed"). For them, RENAR

Core or nothing is enough.

> *A quantitative ROI model (per-project savings order, etc.) currently lives in the research materials; porting a calibrated version into this section is planned in the **phase-8 backlog** for v1.1.*

10. Related documents

- standard/11-maturity-model — normative definitions of the RENAR-1..RENAR-5 levels (closed list).
- 00-quickstart — a 30-minute sample for a small project.
- 01-walkthrough — a full example on one project.
- 07-failure-modes — what can go wrong during migration; organizational failure patterns §5.
- reference/02-schemas — frontmatter schemas, mandatory for RENAR-2+.
- 03-tool-guide-git — a substrate-specific guide for git.
- 04-document-store-substrate — an informative overview of a document-oriented store substrate.

11. Decisions fixed for v1.0

- **Legacy backfill bulk-import — bypass is forbidden.** On initial import, artifacts are entered with the status `imported-legacy` (a substrate-specific marker, not part of the normative closed-list lifecycle §10.5–§10.8) and do not participate in conformance assessment until promoted through the normal QG-0 flow. This preserves the §1.7.3 declared-weaker prohibition: validation is not bypassed, only deferred until the moment the artifact begins to claim conformance.
- **A rollback from RENAR-3 → RENAR-2 is permitted** through a formal downgrade per §13.8.2: a new version of the manifest is issued with a lowered `level`. A recovery plan is **not** mandatory (a downgrade is intentional, not a loss of conformance), but it is RECOMMENDED to record the reason for the downgrade in the audit log.
- **Cross-org migration: each subsystem — its own manifest with its own `level`** (§13.4). The organization-aggregate "RENAR-N" is informally defined as the minimum of the subsystems' levels; an organizational manifest is **not** standardized by RENAR v1.0.

11.1 Deferred to v1.1 (phase-8 backlog)

- **Migration templates for teams of 50+ engineers.** The guide targets groups of 5–15 people; a larger scale needs field observations. Owners: RENAR standard maintenance and early adopters.

03. Substrate: VCS — git

> *A concrete implementation of RENAR on git. The `.req` repository structure, submodule pinning between `.req` and `.src`, the PR/MR review workflow, pre-commit hooks for capabilities V1-V6, and the delta-TZ workflow. This guide is informative; the normative content (capability requirements, schemas, lifecycle) lives in `standard/`. For an alternative on a document-oriented store, see guide/04-document-store-substrate.*
>
> ***Prerequisites:*** *standard/03-substrate-versioning (the normative capabilities V1-V6), reference/02-schemas (frontmatter schemas).*

1. When to choose git as the substrate

Git is the default substrate for:

- Open standards and projects (no dependency on internal infrastructure).
- External clients with no dedicated document-store infrastructure.
- Teams with an established PR workflow.
- Projects where requirements and code share one ecosystem (the same VCS provider).

Git is **not** optimal when:

- You need frequent concurrent edits to a single artifact by several authors without merge conflicts.
- You need built-in full-text search without external tools.
- You need a UI for non-technical stakeholders (PMs, legal) without the git CLI.

In such cases, consider a document-oriented store — guide/04.

2. Two-repository layout

The canonical structure is two linked repositories.

```
<project>/
├── <project>.req/                ← REQUIREMENTS repository (a separate VCS repo)
│   ├── tz/                       ← TZ-YYYY-NNN.md (immutable after registration)
│   ├── actz/                     ← ACTZ-NNN.md (TZ clarification protocols, dual
signature)
│   ├── adapt/                    ← ADAPT-NN.md (bridge artifacts)
│   ├── ar/                       ← AR-NNN.md (adversarial-review records,
§7.4.6)
│   ├── br/                       ← BR-NN.md
│   ├── sr/                       ← SR-NN.md
│   ├── specs/                    ← SPEC-* by subfolder (arch/, api/, data/,
int/, …)
│   │   ├── arch/  api/  data/  ui/  ai/  int/  proc/  sec/  ops/  test/  doc/
│   ├── tr/                         ← TR-NN.md (task requirements)
│   ├── tests/                      ← TC-NN.md (test cases; `TC` are standalone
```

```
artifacts)
│   ├── at/                       ← AT-NN.md (acceptance tests from the effective
TZ)
│   ├── dpia/                     ← DPIA-NN.md (optional, for regulated projects)
│   ├── library/                  ← templates, patterns
│   ├── docs/                     ← AI-generated documentation
│   ├── COVERAGE.md               ← auto-generated (`[coverage]` commits)
│   ├── REQUIREMENTS.md           ← auto-generated index
│   └── TEST-PLAN.md              ← auto-generated
└── <project>.src/                ← IMPLEMENTATION repository
    ├── src/                      ← code
    ├── tests/                    ← TC implementations (addressed by
`automation.location`)
    ├── requirements/             ← submodule → <project>.req @ <commit>
    ├── .gitmodules
    └── README.md
```

In `<project>.req/.gitattributes`, for bot-generated artifacts:

```
COVERAGE.md       linguist-generated=true
REQUIREMENTS.md   linguist-generated=true
TEST-PLAN.md      linguist-generated=true
docs/**           linguist-generated=true
```

This excludes them from code statistics and substantially shrinks the PR diff.

3. Capability mapping V1-V6 onto git

Capability (standard/03 §3.3)	Norm	Git mechanism
V1 — immutable history	Past artifact states cannot be rewritten retroactively	Immutable commit history; protected branch + force-push ban on `main`; the `id:` in frontmatter is stable
V2 — atomic change unit	A change applies as a whole or not at all	An atomic commit / squash-merge PR as a single unit; a delta-ADAPT = one PR
V3 — diff & review	A proposed change is reviewed before approval	`git diff` + PR/MR review with a mandatory approve before merge
V4 — branching / change-set	A draft is kept separate from the approved truth	Feature branches (`draft` / `review`) against `main` (`approved`); a PR is a change-set
V5 — cross-substrate version pin	The implementation references a specific version of a requirement	Submodule pin between `.req` and `.src`; commit SHA / `requirement-version` in `verifies[]`
V6 — author & timestamp	Every change unit records its author and time	Commit metadata: author + date; for an AI agent — `ai-provenance`

> *RENAR checks on git (schema validation, status-transition control, coverage reports, link integrity, reconciliation / drift detection) are **enforcement** mechanisms layered on top of the capabilities, not V1–V6 themselves. Their split across pre-commit / CI is §3.1 and §8 of this guide; the normative drift classes are standard/04 §4.11.*

> ***State of these scenarios.** The mechanisms described below for `git` are the **v1.0 target picture**. In practice only `scripts/validate-frontmatter.js` is ready; the rest (`validate-lifecycle`, `validate-references`, `generate-coverage`, `detect-drift`, and others) are in the **phase-8 backlog** (section §8 of this guide). Until the scripts exist, the listed capabilities are upheld manually and through code review; automatic enforcement of RENAR-3+ levels "out of the box" on `git` is not yet achievable. The same caveat applies to the document store (guide/04).*

---

4. Submodule pinning

`<project>.src` pins a **specific commit** of `<project>.req` through a git submodule.

4.1 How it works

- In `<project>.src`, the `requirements/` directory is a submodule on `<project>.req`.
- At build / CI time the code knows: "I implement the requirements as of commit `abc1234`."
- A developer working a task opens `requirements/sr/SR-05.md` via an ordinary `cat` or the IDE — it is a file in the worktree.

4.2 Bump pattern

When the requirements are updated:

1. A PR into `<project>.req` with the requirement changes → review → merge.
2. A **separate** PR into `<project>.src` that **only** moves the submodule pointer:

```
cd requirements
git pull origin main
cd ..
git add requirements
git commit -m "bump requirements: TZ-2026-042 delta + 3 new SR"
```

3. This PR makes it explicit: "the requirements were updated to commit X."

4.3 Why submodule, not subtree / monorepo

- **Provenance:** for any code commit, the exact requirement version it implemented is known.
- **Review isolation:** requirement review (in `.req`) and code review (in `.src`) do not get mixed.
- **delta-TZ atomicity:** a delta is an atomic PR into `.req` plus a subsequent submodule bump in `.src`. There is no "partially applied delta."

- **Document-store compatibility:** when migrating to a document-oriented store, submodule pinning becomes revision-token pinning — the concept is the same.

Alternatives (subtree, monorepo): consider them only if a submodule does not work for reasons specific to your VCS provider; in all other cases submodule is the recommendation.

---

5. PR/MR review workflow

Two review levels, split across the repositories.

5.1 Review in `<project>.req`

Reviewer focus:

- frontmatter is schema-valid.
- The citation to the TZ / ADAPT is present and valid.
- The `parent` / `verified-by` / `constrained-by` links exist.
- The lifecycle transition is legitimate.
- A source citation is present in the body for every normative statement (at RENAR-4+).
- The adversarial AI-review prompt passed (at RENAR-5).

Approval = QG-0 (standard/10 §10.3.1).

5.2 Review in `<project>.src`

Reviewer focus:

- The submodule pointer matches a merged commit in `.req`.
- The implementation references current SR / SPEC through `verifies[].requirement-version`.
- New / changed TC match the contract in `.req`.
- No changes to TC pass/fail criteria without a `[test-spec-change]` tag.

Approval = QG-2 (standard/10 §10.3.3) — after the automated TC pass.

5.3 Forbidden anti-patterns

- **A PR spanning both `.req` and `.src`** — it breaks review isolation; the substrate hook forbids it.
- **Changing the submodule pointer without a merged commit in `.req`** — the pointer references an untracked commit; CI blocks it.
- **`[test-spec-change]` without a separate approval** — a TC pass/fail criterion changed together with a code fix; CI blocks the merge.

---

6. Pre-commit and pre-merge hooks

The minimal substrate hook set for RENAR-3+ on git.

6.1 Pre-commit (in `<project>.req`)

```
# Runs on commit into .req
# Capability V2: schema validation
yamllint --strict $(git diff --cached --name-only | grep '\.md$')
node scripts/validate-frontmatter.js $(git diff --cached --name-only --diff-
filter=AM)

# Capability V3: legal lifecycle transitions
node scripts/validate-lifecycle.js $(git diff --cached --name-only --diff-
filter=M)

# Capability V5: reference integrity (fast check of changed files only)
node scripts/validate-references.js --changed-only $(git diff --cached --name-
only)
```

6.2 Pre-merge (CI job in `<project>.req`)

The full checks that are too slow for pre-commit:

```
- name: Full reference validation (V5)
  run: node scripts/validate-references.js --all

- name: Coverage report regeneration (V4)
  run: node scripts/generate-coverage.js
  # commits as [coverage] bot user

- name: Drift detection (reconciliation, weekly)
  run: node scripts/detect-drift.js
  if: github.event.schedule == '0 0 * * 0'
```

6.3 Pre-merge (CI job in `<project>.src`)

```
- name: Submodule points to merged commit
  run: |
    cd requirements
    git fetch origin
    git merge-base --is-ancestor HEAD origin/main

- name: TC versions pinned to current requirement-version
  run: node scripts/validate-tc-version-pinning.js

- name: No Pass/Fail change without [test-spec-change]
  run: node scripts/validate-test-spec-changes.js
```

7. TZ workflow on git

7.1 Forward workflow (project from scratch)

Initial creation of the requirements axis from a new TZ:

1. **branch in `.req`:** `git checkout -b init/TZ-2026-001` in `<project>.req`.
2. **TZ registration:** `tz/TZ-2026-001.md` (immutable after registration; record the client signature and date).
3. **Adversarial review of the TZ and — on findings — the ADAPT:** the review is always mandatory and its verdict is recorded as `ar/AR-001.md` (§7.4.1.2). On a "findings present" verdict, `adapt/ADAPT-001.md` is created — Forward (an interpretation per § of the TZ) + Backward (findings), standard/07; the ADAPT is an internal document and is not put to the client. On a "no findings" verdict no ADAPT is created, and BR / SR / SPEC carry `source.tz-section` + `source.adversarial-review-ref: AR-001` (§7.4.1.1).
4. **ACTZ:** findings that require a decision from the client are put forward as the protocol `actz/ACTZ-001.md` — "TZ Clarification Protocol No. 1" (§7.13). `status: draft → sent → signed`; the client's and the contractor's signatures are recorded with author + timestamp (V6). Once signed, every finding receives `decided-in: ACTZ-001 §M`.
5. **Architect's signature → QG-ADAPT-approve:** ADAPT moves to `approved` (the Architect's signature alone, §7.5), `open-questions-count == 0`.
6. **Decomposition:** the AI agent derives BR from the approved ADAPT, then SR (`source.adapt`) and SPEC (`source.adapt`; SR references them through `constrained-by[]`, §8.6.1), plus paired pos/neg TC.
7. **QG-0 approval** of each artifact (`draft → approved`); a CI dry-run of the new TC.
8. **PR into `.req` → merge;** the bot regenerates REQUIREMENTS.md / COVERAGE.md / TEST-PLAN.md.
9. **Set up `.src`:** add the submodule `requirements/ → <project>.req @ <commit>`, create TR, implement, QG-2.
10. **AT before the trials:** an isolated agent (a different model, with no access to `.src` or to the internal artifacts of `.req`) derives `at/AT-NN.md` from the effective TZ — `tz/` plus the signed `actz/` (§7.14). The run refreshes `ACCEPTANCE.md`; the release gate requires every AT to be green (§10.4.3).

7.2 Delta-TZ workflow

The full sequence for applying a new delta-TZ.

1. **branch in `.req`:** `git checkout -b change/TZ-2026-042` in `<project>.req`.
2. **delta-TZ creation, adversarial review, and — on findings — the delta-ADAPT:** `tz/TZ-2026-042-delta.md` (the delta text); the adversarial reviewer issues a verdict, recorded as `ar/AR-NNN.md`. On a "findings present" verdict, `adapt/ADAPT-NNN-delta.md` is created (forward interpretation + backward findings, standard/07 §7.6); on a "no findings" verdict no delta-ADAPT is created and the affected BR / SR / SPEC reference `source.tz-section: TZ-`

`2026-042-delta` directly (§7.6.1). The client's decisions on the delta-TZ findings are recorded in a new `actz/ACTZ-NNN.md` (dual signature); the delta-ADAPT is approved by the Architect's signature before the affected requirements are modified. The effective TZ is recomputed — and the ATs are regenerated from it.

3. **Impact analysis:** the AI agent finds the affected BR / SR / SPEC / TC; marks the TC as `obsolete-pending`.
4. **Update artifacts:** the AI agent updates / creates BR / SR / SPEC and paired TC (pos+neg) on the same branch.
5. **Adversarial critic review:** at RENAR-5 — mandatory (a different AI model).
6. **CI dry-run:** the new TC must run without infrastructure errors.
7. **Finalize:** version++, status: `approved`, regenerate REQUIREMENTS.md / COVERAGE.md / TEST-PLAN.md (bot).
8. **PR into `.req` → QG-0 approval → merge.**
9. **branch in `.src`:** `git checkout -b change/TZ-2026-042` in `<project>.src`.
10. **Bump submodule:** `cd requirements && git pull && cd ..`.
11. **Create TR / tasks:** for new / changed SR (via `/task` or substrate-specific tooling).
12. **PR into `.src` "bump requirements + tests + new tasks" → review → merge.**
13. **Development:** pick up a TR → implement → CI → the bot fills `last-run` in the TC.
14. **QG-2:** on green TC — approval → the AI agent moves the requirement to `verified`.

Every step except (1) and (9) is automated natively for the substrate through scripts or CI.

---

8. Migration notes: scripts/ modernization

The existing scripts/ are bash + node helpers from an early era of the project. Modernization status as of v1.0:

- ✅ **Legacy terms cleaned up.** `req-branch.sh`, `req-finalize.sh`, `req-ai-instructions.md`, `req-use-template.sh` no longer use the deprecated `INT-SR`, `AIC`, `UIC`, `tech-specs`, `ai-concepts`, `ui-concepts`. The closed list of current types is standard/04 §4.4 (BR/SR/TR + 11 SPEC). Residual mentions in reference/01-glossary.md and reference/02-schemas.md are legacy mappings for projects migrating from earlier versions.
- ✅ **Schema validator created.** `scripts/validate-frontmatter.js` — a Node ES module that checks the frontmatter of every `.md` in `standard/`, `guide/`, `reference/`, `core/`: required fields `title` / `order` / `lang ∈ {ru, en}`. Run: `node scripts/validate-frontmatter.js [--quiet]`; exit 0 if all are valid, exit 1 on the first error. The schema source is reference/02-schemas.
- ✅ **ADAPT in the forward workflow.** Initial creation (TZ → adversarial review → an ADAPT on findings → BR) is documented in §7.1 of this guide; the delta workflow with the delta-ADAPT is in §7.2, per standard/07 §7.6.
- ⏳ **Pre-commit hooks** (§6.1) are still partly scattered across `req-finalize.sh`. The target state is to factor them into separate `scripts/validate-*.js`; `validate-frontmatter.js` is the first such module.

This chapter describes the **target script set** at the v1.0 release; lines without a ✅ mark are work for **phase 8** (continuation of the backlog).

9. Common operations

Shortcuts for frequent operations.

Operation	Command
Find an SR by ID	`grep -r "^id: SR-12" sr/`
Find the TC verifying SR-12	`grep -rl "id: SR-12" tests/`
Find an orphan SR (no TC)	`node scripts/find-orphans.js sr`
Create a new SR from a template	`cp library/templates/sr.md sr/SR-NN.md && $EDITOR sr/SR-NN.md`
Diff frontmatter over a period	`git log --all --since=... -- sr/`
Current requirement-version of SR-12	`yq '.version' sr/SR-12.md`
List stale TC (last-run < current version)	`node scripts/list-stale-tc.js`

10. CI/CD integration patterns

10.1 Bot-commit conventions

Auto-generated artifacts are committed by substrate hooks with conventional commit-message tags for machine parsing:

Tag	When	What is committed
`[coverage]`	Post-merge in `.req`	Regeneration of COVERAGE.md / REQUIREMENTS.md / TEST-PLAN.md
`[baseline-update]`	After approval of a PR that changes a baseline	Regeneration of PNG baselines for SPEC-UI
`[bump-req]`	Submodule bump in `.src`	Only the submodule pointer + minimal metadata
`[reconcile]`	The reconciliation hook finds drift	Auto-fix of obvious mismatches; flag for the rest
`[test-spec-change]`	A TC pass/fail criterion changed	Manual; requires a separate approval from a reviewer

The substrate hook parses the commit message and applies different validation rules depending on the tag. `[coverage]` / `[bump-req]` / `[reconcile]` commits come from the bot user; `[baseline-update]` / `[test-spec-change]` come from a human + bot signature.

10.2 Bot-user setup

- A separate bot account (e.g. `renar-bot@<org>`) with **write** permission in `<project>.req` and `<project>.src`.
- Bot commits are signed (GPG / SSH commit signature).
- The bot user **cannot** approve a PR (separation of duties — approval is always human).

10.3 branch protection

In both repositories:

- `main` (or `master`) is protected. A push requires a merged PR.
- Required status checks: schema validation (V2), reference integrity (V5), lifecycle validation (V3).
- Reviewers required: ≥ 1 for most; ≥ 2 for priority=must BR / SR / SPEC.

---

11. Cross-references

- standard/03-substrate-versioning — the normative substrate requirements (capabilities V1-V6).
- reference/02-schemas — frontmatter schemas for the validation hooks.
- 02-transition-guide — where this guide fits into the path from pre-RENAR to RENAR-N.
- 04-document-store-substrate — an informative overview of the document-oriented store (the git alternative).
- 07-failure-modes — what can go wrong on git as the substrate (the hook-bypass scheme, etc.).
- standard/10-lifecycle-qg — the normative lifecycle transitions that the validate-lifecycle hook must enforce.

---

12. Open questions

- Should the minimal hook implementations be standardized as a **single** specification description that both VCS and the document store rely on?
- Submodule vs subtree: which conditions make subtree an acceptable alternative? The guide is currently unambiguously in favor of submodule.
- `[coverage]` bot user: best practice for signing / permissions in multi-org projects?
- Drift-detection cadence: weekly is a good default, but what about high-velocity teams (> 50 PR/week)?

---

04. Substrate: document-oriented store (overview)

> ***Informative appendix.*** *The normative V1–V6 capabilities are in standard/03. A practical example on a distributed VCS (git) is in guide/03.*

A **document-oriented store** is a substrate where RENAR artifacts are stored as versioned documents: a stable identifier, a chain of revisions (revision token), atomic update, and an API gateway for lifecycle transitions and signatures.

RENAR is formally **not tied** to the document-store class of systems — only the capabilities (**V1–V6**) are normatively prescribed (§3.3).

1. When to choose a document-oriented store

A good fit if:

- you need a centralized enterprise requirements store across several projects;
- non-technical stakeholders work through a web UI rather than through a VCS;
- federation of cross-project links and search is a **key** product requirement;

A poor fit if:

- the team is already standardized on a git workflow with PR/MR review;
- there are no resources to maintain a document-store server + search index + API gateway;
- you need **fully-fledged** test cases (`TC`) as objects in the same environment without separately configuring custom document types (see §9 — the presence of `TC` is required for declared RENAR conformance);

2. Mapping the V1–V6 capabilities (generalized)

Capability	Typical document-store mechanism
V1 — immutable history	A chain of immutable document revisions + a stable `_id`
V2 — atomic unit of change	A single whole document-version increment per step
V3 — comparison and review (diff)	An approval flow through the API and the UI; interception of direct status edits
V4 — branching and merging	Draft documents or conflict branches (depending on product capabilities)
V5 — version pinning	A field referencing the revision in linked artifacts
V6 — author and timestamp	Document fields `author` + `updated_at` per revision

A comparison with a distributed VCS is in §3.4 (an illustrative table; not a normative contract).

3. Migration VCS ↔ document store

Migration is possible if both substrate implementations preserve:

- the canonical artifact identifiers (`BR` / `SR` / `SPEC` / `ADAPT`);
- the traceability links;
- the lifecycle states from the closed list in §10.

The migration procedure is a project-specific runbook; the normative minimum is a check of V1–V6 before cutover (§3.8).

4. See also

- 03-tool-guide-git.md — the substrate-specific guide for a distributed VCS (git)
- 03-substrate-versioning.md — the normative V1–V6
- 02-schemas.md — canonical fields; the substrate-native projection is informative

05. Comparison with SAFe

> *Mapping RENAR (the standard for **requirements**) onto SAFe 6.0 (the standard for **scaled agile coordination**). The documents are compatible but have a different scope: SAFe regulates how teams coordinate work at scale; RENAR regulates what a requirement is and how it is verified. This chapter is for teams that already run on SAFe (an enterprise multi-team ART is the typical case) and want to keep their SAFe ceremonies while adding RENAR artifacts as the primary Source of Truth for requirements.*
>
> ***Prerequisites:** familiarity with RENAR Core (the 5 rules, ADAPT, the 2 QGs) and basic SAFe 6.0 terminology (Epic / Capability / Feature / Story, WSJF, PI, ART).*

1. Scope: what RENAR regulates, what SAFe regulates

RENAR and SAFe overlap at the level of *work artifacts* (Feature, Story) but solve different problems.

Aspect	SAFe 6.0	RENAR
Standard type	Scaled Agile coordination framework	Requirements-engineering standard
Primary artifact	Epic / Capability / Feature / Story	TZ → ADAPT → BR → SR → SPEC → TC
What it regulates	Cadence, roles, ceremonies, flow	Schema, lifecycle, verifiability, drift control
Artifact substrate	Choice of task-management tool (Jira / Rally / ADO / other)	Substrate-agnostic; normatively — the **V1–V6** capabilities (glossary §2.7)
Quality checkpoints	Definition of Done at the Feature level	`QG-0` (readiness to start) + `QG-2` (verified)
AI-nativeness	Does not regulate how to work with AI	AI as a full participant (adversarial expertise, evaluating scenarios, `judge` models)

Key principle: SAFe and RENAR are compatible. SAFe says "how to coordinate teams on an ART", RENAR says "what must be true about each requirement for it to count as `verified`". A Feature in SAFe ≡ an SR in RENAR — this is **one and the same artifact**, described from different angles.

2. Master mapping table

SAFe artifact	RENAR artifact	Where it lives	Owner	Acceptance criteria
Strategic Theme	(outside RENAR scope — corporate strategy)	strategic docs	Executive	business level

Portfolio Epic	A group of BR for one system	`<system>.req/br/` aggregated	Lean Portfolio Mgmt	All BR in the group with status `verified`
Capability	Subsystem BR (if the subsystem has its own stakeholder)	`<subsystem>.req/br/`	Solution Architect	All subsystem BR `verified`
Program Epic	Optionally — a large initiative inside the ART, aggregated SR	`<system>.req/sr/` aggregated	Product Manager	The target outcome metric achieved
Feature	SR (or a linked group of SR)	`<subsystem>.req/sr/`	Product Owner	TC from `verified-by` green on the current `requirement-version` (QG-2)
Story	TR (Task Requirement)	`<subsystem>.req/tr/` or task tracker (Jira, Linear, GitLab Issues)	Team	All AC met, evidence recorded, code merged
Enabler Epic	SPEC-AI / SPEC-ARCH / SPEC-OPS	`<system>.req/specs/`	Architect	Eval metrics within thresholds; baseline recorded
Spike	A research TR + a Decision in the decision log	`<subsystem>.req/tr/` + decision log	Team	Decision recorded and linked to the originating SR
Defect	TR with `parent: SR-NN` (the violated SR)	task tracker + auto-derived `children[]` in the SR	Team	Negative TC passes, regression test added

Forbidden mappings: INT-SR is a deprecated term (§4.14.1 Terms), replaced by `SR` with `constrained-by: [SPEC-INT-N]`. See §6 below.

3. WSJF prioritization for RENAR

SAFe uses **WSJF (Weighted Shortest Job First)** as its prioritization framework. RENAR adapts it for the BR / SR level.

3.1 Formula

```
WSJF = (User-Business Value + Time Criticality + Risk Reduction & Opportunity Enablement) / Job Size
```

Each component is rated on a 1-20 scale (modified Fibonacci); WSJF is a relative metric and is only meaningful when comparing BR/SR within a single backlog.

3.2 BR frontmatter extension

```yaml
---
id: BR-12
title: "Reduce employee registration time to < 2 minutes"
status: approved
priority: must
prioritization:
  framework: WSJF
  components:
    user-business-value: 10
    time-criticality: 8
    risk-reduction-opportunity: 6
    job-size: 5
  wsjf-score: 4.8               # auto-calculated: (10+8+6)/5
  prioritized-at: 2026-05-03
  prioritized-by: "@product-owner"
---
```

The `prioritization` field is OPTIONAL in Core RENAR and REQUIRED on an ART that applies SAFe.

3.3 When it applies

- **Mandatory** for BR with `priority: must` in projects coordinated through an ART.
- **Recommended** for all BR in a backlog that goes through PI Planning.
- **Not applied** for SR — an SR inherits the priority of its parent BR. Reshuffling SR within a single BR is the Product Owner's decomposition task, not WSJF.

3.4 Alternatives

If a team does not use SAFe / WSJF, RENAR permits other frameworks:

- **MoSCoW** (Must / Should / Could / Won't) — the simplest one, for small projects. Already present as the `priority` enum in RENAR Core.
- **RICE** (Reach × Impact × Confidence / Effort) — for product-driven teams (typically B2C).

RENAR regulates **the field and its schema**; the choice of framework is up to the project. See also reference/02-schemas.md for the permitted values of `prioritization.framework`.

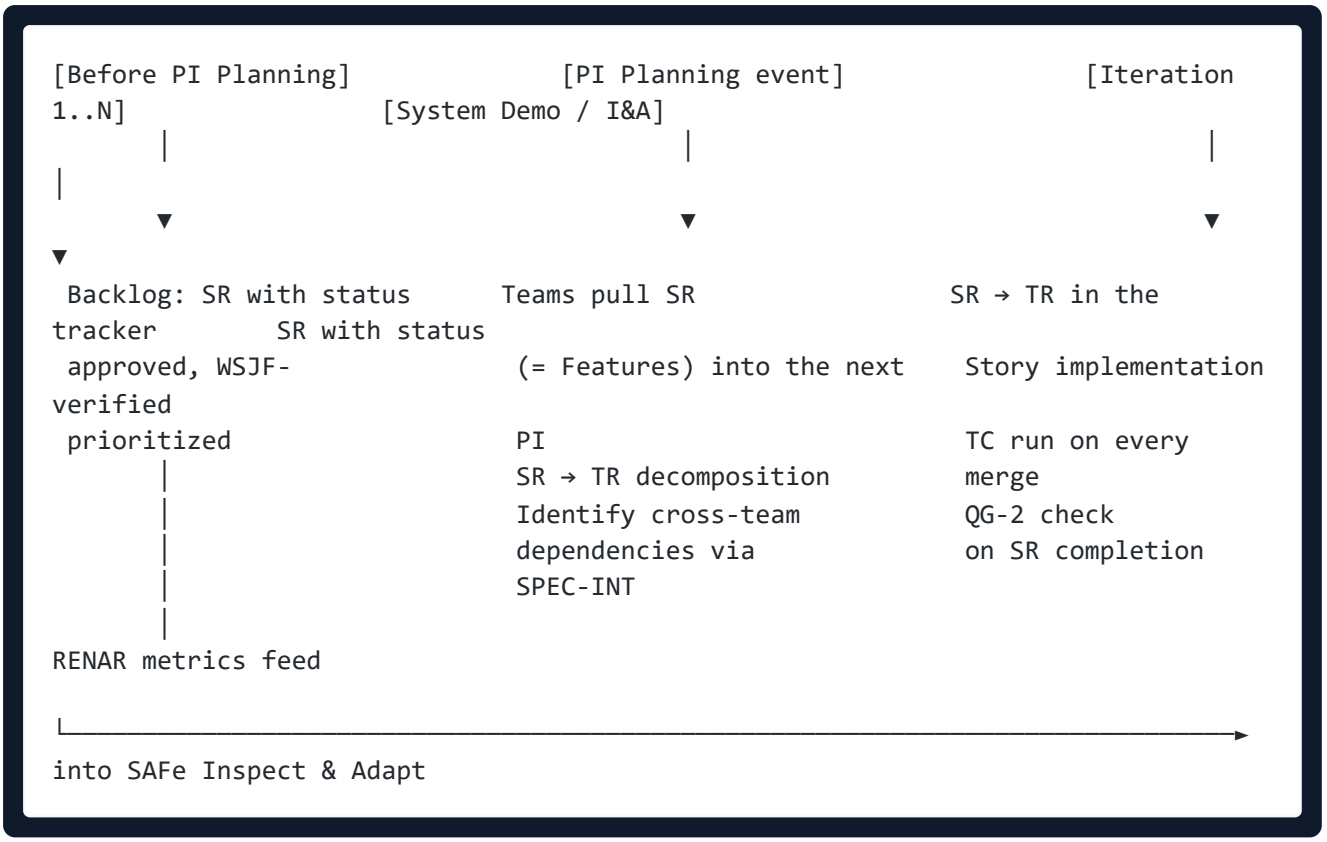
---

4. PI Planning integration

4.1 What a PI is

A **Program Increment (PI)** is a fixed time box (usually 8-12 weeks) in SAFe during which an ART commits to a set of Features from the shared backlog. PI Planning is a two- or three-day event before each PI, where teams jointly fix their commitment.

4.2 Where RENAR artifacts enter the PI flow



4.3 RENAR artifacts in each ceremony

SAFe ceremony	RENAR input	RENAR output
Backlog refinement	BR / SR with status `draft` or `approved`	A refined ADAPT, an updated WSJF
PI Planning	WSJF-sorted SR backlog, ADAPT docs	Commitment to SR in the PI; SPEC-INT for cross-team
Iteration Planning	SR + decomposed TR	TR in `approved`
System Demo	SR with status `verified`	Evidence from the TC last-run
Inspect & Adapt	RDLT, Coverage Velocity, Hallucination Rate metrics	Backlog / process adjustments

5. ART coordination and roles

5.1 SAFe roles ↔ RENAR responsibilities

SAFe role	RENAR responsibility

RTE (Release Train Engineer)	Coordinates cross-team dependencies via SPEC-INT; owner of the PI Objectives ↔ RENAR metrics mapping
Product Manager	Owner of portfolio Epics = groups of BR at the system level
Product Owner	Owner of Features = SR at the team level; accountable for QG-0 (readiness to start) and QG-2 (verified)
System Architect	Owner of SPEC-* (especially SPEC-ARCH, SPEC-INT); consulted during BR → SR decomposition
Tech Lead	Accountable for QG-2 at the SR level; owns the team's TR backlog
Business Owner	Approver of BR / ADAPT from the business side
Solution Architect	Owner of BR at the subsystem level (when the subsystem has its own stakeholder)
Scrum Master	Owner of ceremonies; does not own RENAR artifacts directly

5.2 Cross-team coordination

In a typical SAFe organization with several teams on a single ART:

- **Each team** owns its SR / TR in `<subsystem>.req/` .
- **The RTE / Solution Architect** own the SPEC-INT in `<system>.req/specs/int/` — the shared integration contracts between subsystems.
- **Changing a SPEC-INT** requires cross-team agreement (QG-0 from every affected team).

Rule: ART-level roles (the RTE) **do not edit** SR in subsystems directly. Coordination flows through SPEC-INT, which is explicitly `constrained-by` for every affected SR.

---

6. Cross-team dependencies via SPEC-INT

When a Feature in one subsystem blocks / depends on another:

6.1 An SR with a dependency

```yaml
# In <subsystem-a>.req/sr/SR-05.md
---
id: SR-05
parent: BR-12
title: "User registration via corporate email"
status: approved
constrained-by:
  - id: SPEC-INT-01
    ref: "specs/int/SPEC-INT-01-auth-handshake.md"
    requirement-version: "1.2"
verified-by:
  - TC-23
```

```
  - TC-24
---
```

6.2 SPEC-INT as a contract

```
# In <system>.req/specs/int/SPEC-INT-01-auth-handshake.md
---
id: SPEC-INT-01
type: SPEC-INT
title: "Auth handshake between Subsystem A and Subsystem B"
version: 1.2
participants:
  - subsystem: "subsystem-a"
    role: "client"
  - subsystem: "subsystem-b"
    role: "provider"
status: approved
verified-by:
  - TC-INT-01    # contract test
---
```

6.3 Breaking changes in SPEC-INT

Any change to `SPEC-INT.version` with breaking semantics (§4.11 Drift classes) requires:

1. An ADAPT-level discussion (why we are changing it).
2. Agreement from every `participant` (QG-0 from each team).
3. A migration plan for existing implementors (how the old SR will stay valid or be adapted).

The RTE **MUST** check SPEC-INT consistency across subsystems regularly (usually once per sprint) — this is part of Inspect & Adapt and feeds into the conformance self-assessment (§13 Conformance).

7. Definition of Done at each level of the hierarchy

DoD is the set of **formal** conditions that are checked automatically (substrate hooks). Each level of the hierarchy has its own DoD.

Level	RENAR artifact	DoD criterion
Strategic Theme	—	(outside RENAR scope)
Portfolio Epic	A group of BR	All BR in the group → `verified`, KPI impact confirmed via an outcome metric
Capability	Subsystem BR	All BR with status `verified`; outcome metric measured
Feature	SR	QG-2 passed: every TC from `verified-by` has `last-run.result = pass` on the current `requirement-version`

Story	TR	All AC met, evidence recorded, code merged, automated test exists
Enabler	SPEC-AI / SPEC-ARCH / SPEC-OPS	Eval runs cleared the thresholds (for SPEC-AI); baseline recorded (for all)

Key point: at the Feature level the DoD in SAFe **coincides** with QG-2 in RENAR. There are not two different "definitions of done" — it is one and the same gate, described from different angles.

8. Built-in Quality ↔ RENAR mechanisms

The SAFe Built-in Quality principle: quality is built into the process, not ad-hoc. RENAR implements Built-in Quality through normative mechanisms:

SAFe Built-in Quality practice	RENAR mechanism
Continuous Integration	Substrate hooks + CI on every change in requirements (§13 Conformance); reconciliation hook (drift detection, §4.11)
Test-First	TC are created **before** implementation; QG-0 requires `verified-by` to be empty only when `status: draft`, not `approved`
Refactoring	The continuous reconciliation hook (§2.4.2); detects drift between the requirements and the code
Pairing / Mobbing	AI generator + AI critic (§5.2 Roles) — pair generation/review as a normative role
Definition of Done	QG-2 as a formal gate, checked automatically (§10 Lifecycle and QG)
Version Control	The **V1** capability (immutable history) without tying to a class of storage environment
Automation	The V2-V6 capabilities — all verifications are automatable

RENAR **does not prescribe** the instrument (Jenkins / GitLab CI / GitHub Actions / Tekton) — only **what** must be checked (the capability), not **how**.

9. RACI of an artifact lifecycle (with SAFe roles)

The full lifecycle of a RENAR artifact with responsibility distributed across SAFe roles.

Activity	Responsible	Accountable	Consulted	Informed
TZ import	AI agent	System Architect	Business Owner	Team, RTE
TZ → ADAPT decomposition	AI generator	System Architect	Stakeholder, AI critic	RTE, Team
Decomposition → BR	AI generator	Product Owner	Business Owner, AI critic	RTE, Team
WSJF prioritization	Product Manager	Product Owner	RTE, Stakeholder	Team

BR → SR decomposition	AI generator	System Architect	AI critic	Team, RTE
SR → SPEC-* decomposition	System Architect	System Architect	AI critic, Tech Lead	Team
TC generation	AI agent	Test Architect	—	Team
QG-0 approval	System Architect	Tech Lead	AI critic	Business Owner, RTE
SR selection for the PI	Product Owner	RTE	Team (capacity)	Stakeholder
SR → TR decomposition	Team	Product Owner	Tech Lead	RTE
TR implementation	Developer	Tech Lead	—	Team
TC run (automated)	Substrate hook	—	—	Team
QG-2 (SR verified)	System Architect	Tech Lead	—	Business Owner, RTE
System Demo	Team + RTE	Product Owner	Stakeholder	RTE, Executive
delta-TZ approval	System Architect + Stakeholder	Product Owner	AI impact analysis, RTE	Team
Spot-check of 5 TC (audit)	System Architect	—	—	RTE
Reconciliation MR	AI reconciler agent	System Architect	—	Team

10. PI Objectives ↔ RENAR metrics

PI Objectives are SMART outcomes for the quarter. RENAR metrics (§12 Metrics, reference/02-schemas.md) feed into PI Objectives:

PI Objective (example)	RENAR metric
"Reduce time from TZ signing to first commit to < 2 days"	**RDLT** (Requirement Decomposition Lead Time)
"Reach Coverage Velocity ≥ 60% per sprint"	**Coverage Velocity** (% of SR with `status: verified` per sprint)
"Lower the Hallucination Rate in new requirements to < 2%"	**Hallucination Rate** (% of AI-generated statements rejected at review)
"0 disputed requirements at acceptance in this PI"	**Dispute Rate at Acceptance**
"Drift detection — all SR consistent with code within 24 hours of merge"	**Reconciliation Findings per Week** (§12.3.10)

This turns RENAR from a "standard for documents" into a **measurable contribution** to ART success. The metrics feed automatically into Inspect & Adapt; the RTE uses them in the retrospective at the close of the PI.

---

11. What to keep from SAFe, what to replace

For teams migrating to RENAR from an already-running SAFe process:

11.1 Keep as is

- **Cadence** (PI, Iterations) — RENAR does not regulate time; use your own rhythm.
- **Ceremonies** (PI Planning, System Demo, I&A, Daily Standup) — RENAR artifacts fit transparently into these events.
- **WSJF** — keep it for prioritizing BR / SR; it fits into `prioritization.framework: WSJF`.
- **ART, RTE, Scrum Master** — the structure and roles are preserved.
- **PI Objectives** — keep them; the mapping onto RENAR metrics (§10) makes them measurable.

11.2 Replace with the RENAR equivalent

- **Feature description in Jira / Rally / ADO** → an SR with full frontmatter in `<subsystem>.req/sr/`. The tracker record becomes a **mirror** of the RENAR artifact, not the primary source.
- **Acceptance criteria in a Feature** → `verified-by: [TC-NN]` with automatically checked TC.
- **Definition of Done on a Feature** → QG-2 (no two different DoDs — it is one gate).
- **Integration agreements between teams** → SPEC-INT (replaces the INT-SR from older SAFe implementations).
- **Architecture Decision Records (ADR)** → SPEC-ARCH / SPEC-AI / SPEC-OPS (one of the 11 SPEC types, §4.4).

11.3 Add (new in RENAR)

- **ADAPT** — the bridge artifact between the TZ and the requirements hierarchy. SAFe has no direct analog; it is created **reactively** (standard/07 §7.4.1) — only when the adversarial reviewer returns a "findings present" verdict. On a "no findings" verdict no ADAPT is created, and BR / SR / SPEC are derived from the TZ directly via `source.tz-section` + `source.adversarial-review-ref`. What is mandatory is the adversarial review itself, not the ADAPT.
- **The AI critic role** — adversarial review of AI generation. Not regulated in SAFe; in RENAR Core it is mandatory for all AI-generated artifacts.
- **Reconciliation (drift detection)** — continuous reconciliation of requirements against the implementation (relies on V5 — end-to-end version pinning). In SAFe it is a manual reconciliation; in RENAR it is a normative mechanism.

---

12. Negative: what this chapter does not claim

- **RENAR is not a replacement for SAFe.** RENAR regulates *requirements*, SAFe regulates *work coordination*. A team can run on RENAR without SAFe (a small project, a single team) or with SAFe (an enterprise multi-team ART scope).
- **RENAR is not a planning standard.** Cadence, capacity planning, velocity tracking — all outside the scope. RENAR says "what must be true about a requirement", not "when the team must finish it".
- **RENAR does not forbid other frameworks.** WSJF, MoSCoW, RICE — all compatible. RENAR fixes **the field** `prioritization.framework`, not the choice of framework.
- **RENAR does not regulate the Jira / Rally / ADO workflow.** A tracker-level implementation is a substrate detail; the normative source is `<subsystem>.req/`.
- **RENAR does not prescribe the PI as mandatory.** A team can run on continuous flow without a PI; in that case sections 4 and 10 of this chapter become informative.

---

13. Resolved decisions for v1.0

- `priority: must` **does NOT require a WSJF score, even in SAFe projects.** Per §12 of this chapter: RENAR fixes `prioritization.framework`, it does not prescribe the choice of framework. `priority: must` is a RENAR MoSCoW marker (reference/02-schemas), independent of WSJF. WSJF is an optimization for SAFe teams, not a normative RENAR requirement.
- **Feature/Capability/Story mapping is substrate-specific.** RENAR regulates the closed list of requirement types (BR/SR/TR + 11 SPEC) and does not introduce SAFe Feature levels. Projects that apply SAFe fix the mapping in a substrate-native `safe-mapping/` manifest (see reference/02-schemas — informative extension).
- **PI Objectives are informative, outside RENAR scope.** The PI is a SAFe coordination artifact; RENAR does not regulate it. If a team wants traceability, an informative `cross-link.pi-objective-id` field in the BR frontmatter is recommended — it is not conformance-gating, but it eases RTE-side queries.
- **Inspect & Adapt: metrics are automated substrate-natively.** Per §10.13 Logging + §12 — the substrate MUST surface COVERAGE / audit-trail substrate-natively. The RTE uses the same data through the query API; manual extraction is an anti-pattern (drift).

13.1 Deferred to v1.1 (phase 8 backlog)

- **An AI heuristic for estimating the WSJF Job Size field** (a 1–20 scale by the number of AC, `TC` complexity, and code surface area). Not regulated in v1.0; specific to the SAFe–RENAR binding. The extended informative mapping — reference/11.

---

14. Relationship to other chapters

- 00-quickstart — the basic RENAR cycle without the SAFe overlay.
- 01-walkthrough — a full example on a small-scale project (without PI Planning).
- reference/01-glossary — the exact semantics of BR / SR / SPEC / TR / TC.
- reference/02-schemas — frontmatter schemas, including `prioritization`.
- standard/04-terms — normative definitions; the mapping onto SAFe is fixed in §4.13.
- standard/08-specifications — the closed list of 11 SPEC types, including SPEC-INT.

---

06. Compliance

> *RENAR is a requirements-engineering standard; it is not a compliance standard in its own right. But RENAR provides the **traceability infrastructure** that makes conformance to other standards (ISO 27001, GDPR, FZ-152, EU AI Act, and others) automatically verifiable. This chapter is a mapping of RENAR artifacts onto 8 key compliance frameworks, plus self-assessment checklists, plus a list of auto-generated artifacts for the auditor.*
>
> ***Prerequisites:** RENAR Core, reference/02-schemas.md, reference/03-ai-risk-register.md.*

1. Compliance principles in RENAR

1.1 Compliance frontmatter. Every requirement with a regulatory rationale MUST carry a `compliance` field in its `frontmatter` — you cannot trace conformance through an external table that is not linked to the artifact. The field is repeatable (one requirement MAY close controls across several standards):

```
compliance:
  - { standard: "ISO 27001:2022", control: "A.5.34", rationale: "Privacy and
protection of PII" }
  - { standard: "GDPR", article: "Art.32", rationale: "Security of processing" }
  - { standard: "FZ-152", article: "Art.19", rationale: "Measures to ensure the
security of personal data" }
```

1.2 Data classification. Every BR/SR that operates on data MUST declare a classification. When `contains-pii: true`, SR-encryption + SR-audit-log + SR-erasure automatically become mandatory.

```
data-classification:
  contains-pii: true              # GDPR / FZ-152 trigger
  contains-financial: false       # PCI-DSS trigger
  contains-health: false          # HIPAA / FZ-152 special categories
  contains-children-data: false   # COPPA trigger
  retention-days: 1095
  data-residency: ["RU", "EU"]
```

1.3 Traceability as the audit foundation. The chain `BR → SR → SPEC → TC → implementation → CI run` is itself the audit trail. The auditor opens the coverage report and sees: which requirements close a control; which TCs verify them; `last-run.date`; `requirement-version`. Audit time: 1–2 days instead of 2–3 weeks. The reconciliation hook (drift detection) guarantees that the evidence has not gone stale between audits.

2. ISO/IEC 27001:2022 — Information Security

ISO 27001:2022 Annex A control	RENAR artifact
A.5.7 Threat intelligence	SPEC-SEC with a threat model; SR with `compliance: A.5.7`
A.5.8 Information security in project management	RENAR itself as process compliance
A.5.34 Privacy and protection of PII	SR with encryption + `data-classification.contains-pii: true`
A.6.3 Information security awareness	The onboarding process includes reading RENAR
A.8.1 User endpoint devices	SR with device requirements (if in scope)
A.8.5 Secure authentication	SR-AUTH-* + SPEC-SEC with an authentication flow
A.8.7 Protection against malware	SR + adversarial review (the AI critic)
A.8.10 Information deletion	SR with deletion logic + `retention-days`
A.8.16 Monitoring activities	SR with logging + SPEC-OPS audit-log
A.8.24 Use of cryptography	SR with an explicit crypto algorithm + ISO 25010 Security
A.8.25 Secure development life cycle	SENAR + RENAR as evidence
A.8.28 Secure coding	TC with security checks; SPEC-SEC with STRIDE coverage
A.12.1.2 Change management (legacy 27001:2013)	Delta-TZ + Impact Analysis (§7.6)

Audit deliverable. Auto-generates a conformance report: scope (BR/SR/TC counts with `compliance.iso27001`) + Coverage by Annex A (control ↔ mapped SR ↔ status). The normative mechanism is capability V4 reporting from versioned artifacts.

3. GDPR (Regulation EU 2016/679)

GDPR Article	RENAR artifact
Art.5 Principles	BR explicitly states the lawful basis
Art.6 Lawfulness	BR with `gdpr.lawful-basis: consent / contract / legal-obligation / vital-interests / public-task / legitimate-interests`
Art.7 Conditions for consent	SR with a consent-management workflow
Art.13 Information to data subject	SPEC-UI with a privacy-notice screen
Art.15 Right of access	SR with export-user-data
Art.16 Right to rectification	SR with edit-user-profile

Art.17 Right to erasure	SR with delete-user-data + propagation to linked entities
Art.18 Right to restriction	SR with suspend-processing
Art.20 Right to data portability	SR with export in a machine-readable format
Art.25 Data protection by design and by default	`data-classification` is mandatory at the BR level
Art.30 Records of processing activities	Auto-generated from BRs with `contains-pii`
Art.32 Security of processing	SR with encryption + access control
Art.33 Notification of personal data breach	SR with a breach-notification workflow + a 72h timer
Art.35 DPIA	For high-risk — a separate `dpia/<feature>.md`

GDPR frontmatter:

```
gdpr:
  data-categories: ["identification: email, name", "contact: phone, address",
"behavioral: usage logs"]
  lawful-basis: contract              # Art.6
  retention-period-days: 1095
  cross-border-transfer: false        # true → SCCs or an adequacy decision are
mandatory
  dpia-required: false                # true → link to the DPIA
  data-subject-rights: [access, rectification, erasure, portability]   #
Art.15/16/17/20
```

DPIA. For high-risk processing — `<system>.req/dpia/DPIA-NN-<slug>.md` with frontmatter (`id`, `type: DPIA`, `gdpr-article: 35`, `related-br[]`, `risks-identified`, `mitigations`) and sections: the processing operation, necessity, risks, mitigation measures, and sign-off with the DPO.

4. FZ-152 "On Personal Data" (RF)

FZ-152 Article	RENAR artifact
Art.5 Processing principles	BR with a stated purpose (`business-context.business-goal`)
Art.6 Processing conditions	BR with a `lawful-basis`
Art.9 Consent to processing	SR with consent management
Art.13.1 Storage within the RF	`data-classification.data-residency: ["RU"]` for Russian clients

Art.14 Right of access to information	SR with export-user-data
Art.15 Right to rectification	SR with edit-user-profile
Art.16 Deletion	SR with delete-user-data
Art.18 Operator obligations	The audit trail via RENAR traceability
Art.18.1 Localization of RF citizens' personal data	`data-residency` is mandatory
Art.19 Protection measures	SR with encryption + access control + ISO 25010 Security
Art.21 Breach notification	SR with a breach-notification workflow
Art.22 Notification to Roskomnadzor	operational, outside RENAR scope

Data residency enforcement. For projects with Russian clients, `data-residency` is mandatory. The substrate hook checks: when `data-residency: ["RU"]`, all upstream SRs (storage, backups, replication) MUST have evidence of storage within the RF; adding a foreign jurisdiction for an RU-resident BR → a block at QG-0.

5. EU AI Act (Regulation EU 2024/1689)

The EU AI Act classifies AI systems by risk level. RENAR requires that the class be stated explicitly:

```
ai-act:
  risk-class: limited                # prohibited | high | limited | minimal
  rationale: "Generative AI for document drafting; no autonomous decisions
affecting users' legal rights"
  high-risk-domain: false            # true → a conformity assessment is
required
  general-purpose-ai: true           # GPAI — Claude / GPT, etc.
```

High-risk AI requirements (Art.9–15) — for the `high` class (financial scoring, hiring, public services, medical diagnostics):

AI Act requirement	RENAR artifact
Art.9 Risk management system	SPEC-AI + AI risk register (reference/03)
Art.10 Data and data governance	`eval-datasets/` with provenance + spot-check
Art.11 Technical documentation	SPEC-AI + ISO/IEC 5338 conformance (§7)
Art.12 Record-keeping	tool_event audit + `ai-provenance`
Art.13 Transparency to users	SPEC-UI with an indication of AI processing
Art.14 Human oversight	One-click approval + spot-check at QG-0 / QG-2
Art.15 Accuracy, robustness, cybersecurity	Eval-tests (`tc-type: eval`) + adversarial review

GPAI obligations (Art.51–55). If a General-Purpose AI Model (Claude, GPT, Gemini) is used: `ai-provenance` in the frontmatter of any AI-generated artifact (model id, version, prompt hash); the model's technical document (if it is a GPAI with systemic risk) — an external document + a reference from SPEC-AI.

6. NIST AI RMF 1.0 (US jurisdiction)

NIST AI RMF Function	RENAR mechanism
Govern	RENAR + SENAR roles + compliance frontmatter
Map	BR with `business-context`, `data-classification`, `ai-act.risk-class`
Measure	Eval-tests with metric thresholds + RENAR metrics (Hallucination Rate, Coverage Velocity)
Manage	Lifecycle with QGs + reconciliation hook + AI risk register

RMF-specific extensions (optional for US projects; not in conflict with ISO/IEC 23894 §7):

```
nist-ai-rmf:
  applicable: true
  govern: { role: "AI Governance Lead", policy-link: "..." }
  map: { use-case-category: "content-generation", affected-stakeholders:
["clients", "internal-team"] }
  measure: { metrics-tracked: ["accuracy", "fairness", "robustness"], baseline-
run-id: "eval-2026-04-01" }
  manage: { risk-tolerance: "low", incident-response-plan: "<link>" }
```

7. ISO/IEC 23894:2023 — AI Risk Management

Guidance on AI risk management. It does not certify, but it provides a structured framework for managing the risks of AI systems. Compatible with NIST AI RMF and the EU AI Act.

ISO/IEC 23894 clause	RENAR artifact
§6.4.1 Risk identification	AI risk register (reference/03)
§6.4.2 Risk analysis	SPEC-AI with a risk-assessment section
§6.4.3 Risk evaluation	A threshold for each risk in `eval-tests`
§6.4.4 Risk treatment	SR-mitigations + post-mitigation evaluation
§6.4.5 Communication and consultation	RACI matrix (05-safe-comparison §9)
§6.4.6 Monitoring and review	Reconciliation hook (drift detection)

Conformance criteria: every AI-generated artifact carries `ai-provenance`; for each AI use case, SPEC-AI documents the identified risks (hallucination, bias, robustness), the mitigations (eval thresholds, adversarial review, human-in-the-loop), and the residual risks; the risk register is updated whenever SPEC-AI changes (reconciliation catches the drift).

8. ISO/IEC 5338:2023 — AI System Life Cycle Processes

An extension to ISO/IEC/IEEE 12207. A convenient framework for AI Act Art.11 technical documentation.

ISO/IEC 5338 process	RENAR stage
Stakeholder needs and requirements definition	TZ + ACTZ (clarification protocols) + ADAPT + BR
Architecture definition	SPEC-ARCH + SPEC-AI
Design definition	SPEC-AI (model card, prompt design, RAG)
Implementation	TR + implementation
Verification	TC (`tc-type: eval` for AI)
Validation	AT — acceptance tests from the effective TZ (§9.19) + spot-check
Operation	Reconciliation hook (drift detection, §4.11)
Maintenance	Delta-TZ + ACTZ + ADAPT iteration
Disposal	Retirement workflow (outside Core RENAR scope)

Conformance evidence: SPEC-AI for each AI use case with a full model card; eval-tests bound to SPEC-AI via `verifies[]`; `ai-provenance` recorded; drift detection active (V6).

9. PCI-DSS v4.0 — Payment Card Industry Data Security Standard

If the project handles payment-card data (CHD):

PCI-DSS v4 requirement	RENAR artifact
Req.1 Network security controls	SPEC-OPS with network segmentation
Req.3 Protect stored CHD	`contains-financial: true` + SR with encryption-at-rest
Req.4 Encrypt CHD transmission	SR with TLS + SPEC-API requirements
Req.6 Develop secure systems	RENAR + SENAR as process compliance
Req.7 Restrict access by need to know	SR with RBAC + SPEC-SEC access-control matrix
Req.8 Authenticate access	SR-AUTH-* + MFA where mandatory
Req.10 Log and monitor access	SR with an audit trail + SPEC-OPS observability

Req.11 Test security regularly	TC `tc-type: security` + pen-test (outside RENAR scope)
Req.12 Information security policy	RENAR + SENAR as a documented policy

CHD scope minimization. The substrate hook checks that new SRs with `contains-financial: true` explicitly justify the need to store CHD — without that justification, the requirement is halted at QG-0.

---

10. Self-assessment checklists

Short yes/no checklists for compliance teams. The full printable kit for a RENAR manifest — reference/08.

ISO 27001:2022 — every BR with `contains-pii: true` has an SR closing A.5.34; the SoA is documented; every in-scope control has ≥1 verifying SR; the coverage report is green; an auditor goes from a control to a passing TC in under 5 minutes.

GDPR — all PII is in the Records of Processing (auto-generated); the lawful basis is stated per processing activity; data-subject rights Art.15–22 are verifiable through SR + TC; a DPIA is performed for high-risk; cross-border transfers are justified (SCCs/adequacy).

FZ-152 — requirements with PII of RF citizens carry `data-residency: ["RU"]`; the substrate hook checks the upstream storage SRs; consent is implemented in an SR with TC evidence; the Art.19 protection measures are covered by an SR with a passing TC.

EU AI Act — every SPEC-AI carries `ai-act.risk-class`; for `high`: Art.9–15 evidence is in the substrate; human oversight at QG-0/QG-2 + spot-check; technical documentation is auto-generated; `ai-provenance` for GPAI components.

NIST AI RMF — all 4 functions (Govern/Map/Measure/Manage) have evidence; an AI Governance Lead is defined in roles; the eval baseline is recorded; an incident-response plan is linked to SPEC-AI.

ISO/IEC 23894 — the AI risk register is filled out and linked to SPEC-AI; every risk has a mitigation in an SR; residual risks are accepted by the owner; V6 is active.

ISO/IEC 5338 — SPEC-AI for each AI use case with a model card; eval-tests bound via `verifies[]`; `ai-provenance` recorded; V6 is active.

PCI-DSS — SRs with `contains-financial: true` have encryption (at-rest + in-transit); the CHD scope is minimized; RBAC is in SPEC-SEC; the audit-trail SR covers all CHD-touching flows; security TCs run regularly.

10.9 RENAR-conformance self-assessment (in an hour)

A project declares its **own** RENAR-conformance level via the `RENAR-CONFORMANCE.yaml` manifest (standard/13 §13.4). Quick checklist (yes/no, single pass):

- The requirements substrate provides all of V1–V6 (§11.4.1 — Notion/Google Docs do not qualify).
- Each TZ has an issued AR (adversarial-review record); every ADAPT produced by a "findings present" verdict is in status approved with the **Architect's signature** (§7.5). There may be no ADAPT at all (a "no findings" verdict) or several (findings at different stages) — both are regular cases.
- A mandatory client signature on the ADAPT is **not** declared: obligations to the client are carried by the ACTZ. Requiring a client signature on the ADAPT is permitted only as `declared-`

`stricter`, never as base conformance (§13.3.3).

- Every finding in status `resolved` carries a `decided-in` pointing at a clause of a **signed** ACTZ; no signed ACTZ is left unreflected in an ADAPT (§7.4.5, §7.13).
- Acceptance runs against the **effective TZ** (the initial TZ + every signed ACTZ, §7.14), not against the ADAPT.
- The ATs were derived by an isolated agent: a model different from the primary agent's, with no access to the ADAPT / BR / SR / SPEC / TC / code; each AT's `tz-version` matches the current revision of the effective TZ; every AT is `passing` before submission (§9.19, §10.4.3).
- No TC counts as evidence without a red history; the exception is a TC of class `implementation-originated`, where a killed mutant serves as the compensation (§9.18.2).
- Requirements of class `implementation-originated` describe internal technical detail only: client-observable behavior is not legalized through this class (§6.13.2).
- All 11 SPEC types are supported natively (a declaration of "type not used" is permitted, "not supported" is not).
- Every normative statement covered by a TC has a paired negative TC.
- QG-0 / QG-1 / QG-2 are declared `required`.
- The manifest contains `renar-version` + `senar-version` + `level` + confirmation of the mandatory clauses (reference/08 §2).
- `assessment-date` is recent, `next-assessment-due` is not overdue.

All 12 are `yes` → the project is conformant to the declared `level`. At least one `no` → the manifest is not issued (§13.5.2).

The four items in the middle of the list are the new mandatory clauses of §13.3.3. They share one point: an obligation to the client cannot arise out of an engineering document the client never read, nor out of the code, nor out of a test written by the same agent and the same model as the implementation.

A filled-in example (RENAR-2, self-assessment) — full schema in §13.4.2:

```
renar-version: "1.0"
senar-version: "1.0"
manifest-version: 1
manifest-id: "CFM-2026-014"
level: "RENAR-2"
level-target: "RENAR-3"
assessment-mode: "self"
assessment-date: "2026-05-20"
assessor: { id: "architect-team-lead", role: "architect", signature-ref: "
<pointer>" }
next-assessment-due: "2026-08-20"
mandatory-clauses-confirmed:
  sot-inversion: true
  substrate-v1-v6: { v1: true, v2: true, v3: true, v4: true, v5: true, v6: true }
  adapt-per-tz: true
  spec-types-closed-list: true
  tc-pos-neg-pairing: true
  quality-gates-closed-list: true
  closed-lists-backward-findings: true
quality-gates: { qg-0: required, qg-1: required, qg-2: required, qg-3: absent,
```

```
qg-4: absent }
spec-types-supported: ["SPEC-ARCH","SPEC-API","SPEC-DATA","SPEC-INT","SPEC-
PROC","SPEC-UI","SPEC-AI","SPEC-SEC","SPEC-OPS","SPEC-TEST","SPEC-DOC"]
exceptions: []
replaced-by: null
```

11. Auto-generated compliance artifacts

Artifacts generated from existing requirements for auditors:

Artifact	Source	Contents
Records of Processing (GDPR Art.30)	BR/SR with `contains-pii: true`	Data categories, lawful basis, retention, recipients
Statement of Applicability (ISO 27001)	BR/SR with `compliance: ISO 27001:2022`	Annex A controls in/out of scope, justification
Data Inventory	All `data-classification` records	Categories, residency, retention, owners
AI System Cards	SPEC-AI	Model card in NIST AI RMF / AI Act format
Audit-trail report	Traceability BR → SR → SPEC → TC → last-run	The chain from control to evidence
DPIA Index	The `dpia/` folder	A list of DPIAs + their DPO sign-off status
AI Risk Register Snapshot	AI risk register	All identified risks + mitigations + residual

Substrate-native reporting: aggregation of evidence from versioned artifacts (V4) into a compliance report on the assessor's request.

Evidence pack — templates (informative; full E2E — guide/09 §E3). GDPR Art.15 trace bundle — a `Control | BR/SR | SPEC | TC | last-run` table + lawful basis + retention + a link to the manifest. FZ-152 Art.14 — a mapping table `Requirement ↔ RENAR artifact ↔ Evidence field`. ISO 29148 trace excerpt — reference/07 (fill in "RENAR frontmatter" for each in-scope SR). Pre-audit self-assessment — reference/08 §2.

12. What RENAR does NOT cover in compliance

RENAR is infrastructure for compliance, but not a **substitute** for: the DPO (Data Protection Officer — a human role with legal accountability); legal review of contracts and privacy notices; pen-testing the implementation; bug bounty / vulnerability management; physical security (data centers, the office); operational security (key rotation, incident response, the monitoring runbook); cyber insurance; regulatory filings (Roskomnadzor notifications, GDPR registration with a DPA, AI Act conformity assessment).

RENAR helps you do compliance **during requirements engineering**. Operational compliance consists of separate processes that *reference* RENAR artifacts as evidence.

13. Resolved decisions for v1.0

- **Compliance frontmatter — conditionally mandatory.** By default it is optional; it is mandatory when `contains-pii: true` (GDPR/FZ-152 trigger) or `contains-phi: true` (HIPAA trigger). An artifact with PII and no compliance frontmatter is non-conformant (§13.3 extensions via a manifest `declared-stricter`).
- **DPIA — mixed model.** The formal DPIA is an external document (legal owns it); the RENAR artifact `dpia/<slug>.md` holds a machine-readable summary + a pointer to the formal document. Separation of concerns while preserving traceability.
- **Data residency — outside RENAR scope.** Per §1.3 (3), tech stack / infra are out of scope. Enforcement is via DevOps-level controls (network policies, cloud regions). RENAR records the **requirement** (`data-classification.data-residency: ["EU"]`), not the enforcement.
- **Custom industry frameworks** (CB RF fintech, HIPAA healthcare, etc.) — separate documents (industry-specific addenda as `guide/06-compliance-<industry>.md` or external documents; they **MAY** be `declared-stricter` on top of RENAR conformance).

Deferred to v1.1 (phase-8 backlog): auto-export of evidence for Schrems II and EU ↔ RF transfers — partially addressed by the `cross-border-transfer` flags and links to SCCs, but full automation is unattainable (which SCCs apply is a lawyers' call). Owners: the legal-tech / adapters pairing.

14. Relationship to other chapters

- 00-quickstart — the basic RENAR cycle without the compliance layer.
- 05-safe-comparison — RACI with SAFe roles (including the DPO as Consulted).
- reference/02-schemas — frontmatter schemas: `compliance`, `data-classification`, `gdpr`, `ai-act`.
- reference/03-ai-risk-register — the AI risk register structure.
- reference/04-ai-style-guide — the AI-provenance style.
- standard/04-terms — SPEC-SEC, SPEC-AI, SPEC-OPS terminology.
- standard/13-conformance — RENAR conformance levels (≠ compliance: RENAR-N assesses the maturity of the *process*, compliance is conformance to *external norms*).

07. Failure modes

> *A systematic survey of every known way a RENAR project can break down: technical drift between artifacts and implementation, AI-specific risks, and (most importantly) organizational patterns where the process exists on paper but does not work. Drift classes are normalized in standard/00 §0.3; AI risks — in reference/03. For each failure mode — symptom, how to detect, how to prevent, how to recover.*
>
> ***Prerequisites:*** *RENAR Core, reference/03-ai-risk-register.md.*

1. Map of failure modes

Three classes of problem:

Class	Where it lives	How it surfaces
Drift	A mismatch between different representations of the same entity (frontmatter ↔ DB, requirement ↔ code, TC ↔ requirement)	Reconciliation hook (drift detection, §4.11)
AI risks	Properties of AI generation (hallucination, bias, injection, model drift)	Adversarial review + eval tests + AI risk register (reference/03)
Organizational	A mismatch between the formal process and the team's real practices	Behavioral signals: the approval pattern, the frequency of disputes, the frequency of bypass

Drift and AI risks are caught by substrate mechanisms. Organizational failures are caught by no substrate; they require human-level process reviews. This chapter covers all three.

2. The 8 drift classes

For each: symptom (how it looks from the outside), detection (how to catch it automatically), prevention (how to avoid it), recovery (what to do once it has happened).

2.1 Schema drift

Symptom: The fields in an artifact's frontmatter diverge from what the substrate expects / supports.

Detection: On every change to an artifact, the substrate validates the frontmatter against the schema (reference/02-schemas.md). On a divergence, integration is blocked (QG-0 fails).

Prevention: The schema is the single source of truth (closed list); it is not edited within the project. Schema changes happen only through a change to the full RENAR Standard.

Recovery: Roll the frontmatter back to a schema-valid state; if the change is genuinely needed, open an RFC for a standard change.

2.2 Lifecycle drift

Symptom: Statuses (`draft` / `approved` / `verified` / `deprecated`) and quality-gate names are understood differently across subsystems or across teams.

Detection: Compare the status transitions in the audit trail against the normative state machine (standard/10-lifecycle-qg). Anomalies (a transition without the corresponding pre-conditions) are flagged.

Prevention: Transitions are performed by the substrate mechanism, not by manual frontmatter edits. Capability V3 (state-machine enforcement).

Recovery: Roll back the illegitimate transition; re-run QG-0 / QG-2 through the correct mechanism.

2.3 Source-of-truth drift

Symptom: The same entity is edited in two places (for example, both in the `.req` directory and in the Jira tracker). The versions diverge.

Detection: Periodic reconciliation between the substrate and the tracker; the diff reveals the divergences.

Prevention: At any moment in time, **exactly one SSoT substrate is chosen** for the project. The tracker is a derived view, not the Source of Truth. The substrate hook blocks tracker-only changes to requirements.

Recovery: Declare one substrate the winner; merge the second into the first; stop editing in the second until migration.

2.4 Implementation drift

Symptom: Code in the implementation references an SR that no longer exists (deprecated, removed, renamed). Or: the SR exists, but the implementation has drifted away from it (the behavior does not conform).

Detection: Reconciliation hook (drift detection):

- Forward: walk from a requirement → find the implementing code → run the TC.
- Backward: walk from the code → find references to SR / TC → check that they exist and are `verified` .

Prevention: Requirement IDs are **immutable** — renaming is forbidden. Deprecated requirements stay in the repository with status `deprecated` ; they are not deleted.

Recovery: Open a delta-TZ that explicitly adopts the current implementation (or, conversely, requires the code be rolled back into conformance with the requirement).

2.5 Terminological drift

Symptom: "Verified", "implemented", "approved" mean different things to different people / teams.

Detection: Code-review checklist: "a term not from the glossary was used?" — a flag. Likewise, the substrate validator checks that the values of enum frontmatter fields come only from the closed list.

Prevention: The glossary is the single source of terms (reference/01-glossary). Each term = exactly one lifecycle state.

Recovery: Audit all project artifacts for the use of out-of-glossary terms; replace them or file an RFC to extend the glossary.

2.6 Order / provenance drift

Symptom: Delta-TZ #2 references an SR that was created in Delta-TZ #1, but application happened in reverse order — the SR did not exist at the moment #2 was applied.

Detection: Delta-TZs are numbered and applied strictly in number order. The substrate hook checks that the upstream delta has already been applied.

Prevention: Delta-TZs cannot be renumbered. Each artifact stores `created-by-order` (the delta-TZ of creation) and `last-modified-by-order` (the last update).

Recovery: Roll back the out-of-order application; re-apply in the correct order.

2.7 TC ↔ requirement provenance drift

Symptom: A TC verifies a requirement, but the requirement has already changed — `last-run.requirement-version` is lower than the requirement's current `version`. The test is green, but it checks outdated behavior.

Detection: The coverage report shows a "Stale" category — TCs with an outdated `last-run.requirement-version`. Reconciliation catches this automatically (via the V5 version pin).

Prevention: A TC has the mandatory field `verifies[].requirement-version` — a pinned version. QG-2 forbids moving a requirement to `verified` if at least one TC in `verified-by` has a stale `last-run`.

Recovery: Re-run the stale TC against the current requirement version; update it if the TC itself is outdated.

2.8 Test-fitting drift

Symptom: An AI agent has a trivial path to turning a failing test green — weaken the pass/fail criterion instead of fixing the code. Without protection, tests drift from "strict checker" to "green void".

Detection: A change to a TC's pass/fail criteria without an explicit `[test-spec-change]` tag is flagged by the substrate. A periodic spot-check of 5 random passing TCs once per sprint.

Prevention:

- An MR / change that modifies pass/fail criteria MUST carry the `[test-spec-change]` tag and a separate Engineer approval (not combined with the approval of the code fix).
- Isolation of the judge model: the production model ≠ the judge model.
- A trending test-fitting drift-rate metric.

Recovery: Restore the old criteria; perform a root-cause analysis — why the AI agent chose greening over a fix; update the prompt / system instructions.

---

3. The 14 AI risks (brief summary)

Full descriptions, mitigations, and owners — in reference/03-ai-risk-register. Here is an operational summary: id, name, severity, the main detection signal.

ID	Name	Severity	Detection signal

AIR-01	Hallucination in AI-generated requirements	High	Hallucination Rate metric > threshold; adversarial critic flags
AIR-02	Prompt injection via a client TZ	High	Suspicious pattern in imports; sandbox violation
AIR-03	Model drift / version change	Medium	diff regression on a model switch; baseline eval failure
AIR-04	Bias in AI requirement generation	Medium	Stakeholder map gaps; missing accessibility/locale considerations
AIR-05	Single-model failure (no diversity)	Medium	All artifacts with one `ai-provenance.model`; no multi-model agreement
AIR-06	Test-fitting / greening tests	High	diff in TC pass/fail without a `[test-spec-change]` tag
AIR-07	Hallucinated citations	Medium-High	Citation validator hook fails
AIR-08	Adversarial inputs in client data	High	Application-level (out-of-scope for RENAR, tracked in SPEC-SEC)
AIR-09	Privacy leakage via AI logs	High	PII in the tool_event audit; redaction skip
AIR-10	Knowledge graph poisoning	Medium	Incorrect edges; circular dependencies in the graph
AIR-11	Reconciliation false-positive overload	Low-Medium	Findings/week trending up without real issues; high dismissal rate
AIR-12	Cost runaway (uncontrolled AI spend)	Medium	Project AI cost approaching the budget cap
AIR-13	A Stakeholder does not understand AI-generated requirements	Medium	Dispute rate at acceptance rising; long approval cycles
AIR-14	Vendor lock-in to a specific LLM provider	Medium	All prompts work only on one provider

The risk matrix and review cadence — reference/03 §5-§2.

3.5 Adversarial review (procedure)

> ***Informative.*** *An operational procedure for WC-13; normative requirements — standard/09 §9.4, standard/13 §13.2 (RENAR-5).*

When mandatory (normative): adversarial review is QG-0 for RENAR-5 (§11.8.1); for `SPEC-SEC` / `SPEC-AI` — an external reviewer at QG-0 (§5); declared-stricter MAY broaden the scope (standard/00 §0.6).

Step	Actor	Artifact	Exit criterion

1. Scope	Architect	A list of TCs + the related SR/SPEC	Each `approved` TC in scope has a `tc-type` and `verified-by[]`
2. Critic pass	AI critic (a separate model/prompt)	A findings log with id, severity, and a reference to the TC/SR	Findings are traceable to a concrete clause §9.x; no "generic" recommendations
3. Triage	Architect + RE Engineer	Disposition: fix / accept / reject	Each finding has an owner + rationale; dismissal without rationale is forbidden (see §4.6)
4. Re-run	AI agent or human	Updated TCs + diff	QG-2 pre-condition: `passing-tests / total-tests` for the scope (§9.10)
5. Audit trail	substrate (V1)	A commit/change unit with the `adversarial-review` tag	provenance: model id, prompt version, findings hash (§10.13)

Approval discipline: the "100%" metrics in §9 are a **target at QG-2**, not a guarantee of product quality. AI-risk severity comes from reference/03, not from an editorial override.

Agent panel (no human reviewers): an informative procedure — §3.5 (steps 1–5); the rubric and severity — reference/03.

---

4. Organizational failure patterns

These problems are not caught by substrate mechanisms — they are behavioral patterns of teams. They typically appear 2–6 months after adopting RENAR.

4.1 ACTZ as a formality

Symptom: The TZ clarification protocol is signed without being read. The questions were put to the client as the agent's raw output — a three-page list from which the client cannot tell what they are signing off on. The backward section of the ADAPT is empty or contains yes/no answers without context.

Sign: An ACTZ signed < 24 hours after the draft was generated; the rate of disputed requirements at acceptance is rising.

Mitigation: What is put to the client is a **prepared** list of decisions, not the agent's raw output (standard/07 §7.13.4): the Architect aggregates and rephrases the questions in the language of obligations. The dual signature belongs to the ACTZ, and to it alone; the ADAPT is signed by the Architect alone (§7.5), and demanding a client signature under it is pointless — that fiction is precisely what produced "signed without reading". The backward section MUST contain ≥ 1 non-rhetorical question. Spot-check ACTZs and ADAPTs in I&A.

4.2 SPEC overload

Symptom: The team creates a SPEC for every task, even when SR + TR are sufficient. The SPEC catalog balloons; every PR updates 5+ SPECs.

Sign: The SPEC / SR ratio > 1.5 (the expected value is < 0.3 for projects of medium complexity).

Mitigation: A pre-review checklist: "is a SPEC needed for this change?" A SPEC is justified only when several SRs share a common constraint. See standard/08-specifications.md §8.2 — when a SPEC is

mandatory.

4.3 Hooks as an obstacle

Symptom: The team routinely bypasses the substrate hooks (`--no-verify` , timestamp manipulation, manual status edits).

Sign: The git log / substrate audit trail shows a frequency of bypass commits; QG-0/QG-2 pass in suspiciously short times.

Mitigation: The root cause is hooks that are too slow / too noisy / too strict. Do not "ban the bypass" — fix the hooks. Treat the bypass frequency as a trending metric — if it rises, run a retro with the team.

4.4 Drift detection without action

Symptom: The reconciliation hook generates drift findings, but no one acts on them. The findings backlog grows; old findings are ignored.

Sign: Findings older than 14 days > 30; resolution rate < 20% / week.

Mitigation: Each drift finding gets an owner and an SLA (resolve / accept / reject within N days). Unresolved findings past the SLA are escalated. Reconciliation without human ownership = noise.

4.5 Tracker as a parallel universe

Symptom: The team lives in Jira / Linear / ADO; the `.req` directory is updated once a week "for the record". The tracker is the real Source of Truth, RENAR is a formal artifact for the audit.

Sign: The diff of `.req` vs the tracker > 30% in any given week; commits to `.req` are rare and batched.

Mitigation: The Source of Truth must reside in the substrate, not be tracker-resident. The tracker is a derived view only. If the team cannot work without the tracker — the substrate must push *into* the tracker, not the other way around.

4.6 Critic burnout

Symptom: The AI critic (adversarial review) generates many findings; gradually the developer / Architect start ignoring its output. Findings are rejected without consideration.

Sign: The AI critic's dismissal rate > 80%; time-to-dismiss < 30 seconds per finding.

Mitigation: Tunable thresholds for the critic. If the false-positive ratio is high — recalibrate the prompt / model. The "critic finding → real issue" metric (the % of dismissed findings that later surfaced as a defect) — if it is 0%, the critic is useless.

4.7 Single-engineer dependence

Symptom: Only one Engineer on the project "understands RENAR". All QG-0 / QG-2 pass through them. If they go on vacation — the process stalls.

Sign: The bus factor of RENAR ownership = 1. The distribution of QG approvals is heavily skewed toward one person.

Mitigation: Paired onboarding (at least 2 Engineers on the project know RENAR). Rotation of the QG-approver role. Documentation of project conventions in `<project>.req/CONVENTIONS.md` .

4.8 Ad-hoc delta

Symptom: Requirement changes happen without a delta-TZ being filed — "let's just change SR-12 right in the repository".

Sign: Direct commits to `<system>.req/sr/*` without a corresponding delta-TZ; the `created-by-order` field is empty.

Mitigation: The substrate hook blocks mutation of existing requirements without a `delta-ref` in the commit metadata. All changes go through the delta-TZ workflow (standard/07-adapt §7.6).

4.9 TC abandonment

Symptom: TCs are created alongside the requirements, but then they are never run. `last-run` is older than N months; the coverage report shows "green" TCs that in reality have not run in half a year.

Sign: Median `last-run` age > 90 days; the TC count grows, the run count does not.

Mitigation: The substrate runs TCs automatically on a schedule (capability V4). A TC without a `last-run` for N days is automatically marked `stale`; QG-2 blocks until they are re-run.

4.10 A test born green

Symptom: A TC written by the same agent, in the same session, as the implementation — and green from birth. Formally there is coverage; in fact the test checks the code, not the requirement.

Sign: The TC's run history contains no red result at all; the TC's provenance coincides with the provenance of the implementation change unit.

Mitigation: Authorship isolation along three axes — time (the TC is frozen in `ready` before the TR starts), author (the test agent ≠ the implementation agent), and change (a Pass/Fail edit only via `[test-spec-change]`), standard/09 §9.18. The pinning run before implementation MUST be red; a green one there is a signal to investigate, not a success. The single exception is a TC of class `implementation-originated`, which by construction has no red history: there its teeth are proven by a killed mutant.

4.11 Acceptance that confirms itself

Symptom: The ATs were generated by an agent that was given access to the SRs and the ADAPT "for context" — or simply by the same model that wrote the implementation. Every AT is green, and acceptance at the client fails.

Sign: The AT's `generator` does not attest isolation; or the AT's `tz-version` lags behind the current revision of the effective TZ.

Mitigation: ATs are derived **only** from the effective TZ, by an isolated agent on a different model, with no access to the internal loop (standard/09 §9.19.2) — otherwise the acceptance level collapses into the verification level and the error of interpretation receives a false confirmation of conformance. ATs are regenerated before **every** round of trials: a program derived at planning time will, by the end of a long engagement, be checking the system against a year-old contract.

---

5. Failure recovery playbook

What to do once the system is already broken. The sequence is common to all failure modes; the specifics depend on the class.

Step 1: Stop the bleeding

Find and halt the ongoing damage:

- Drift: freeze further changes in the affected area.
- AI risk: suspend AI generation for the affected class of artifacts.
- Organizational: take it to a retro / I&A — this is not a technical fix.

Step 2: Quantify

Measure the damage:

- How many artifacts are in a drift state?
- How many releases since the problem arose?
- Which SR / SPEC / TC are affected? (Capability V4 — coverage / drift report)

Step 3: Triage

Segment the damage into:

- **Critical** — already in production, affecting users. Hot-fix.
- **Active** — in the current PI, affecting ongoing work. Block PI exit.
- **Historical** — old artifacts, not actively used. Batch fix.

Step 4: Fix

For each class, the corresponding fix:

- Schema drift → roll back the frontmatter; RFC if the schema needs to be extended.
- Implementation drift → delta-TZ adopt OR roll back the code.
- TC drift → re-run the TC against the current requirement-version.
- Test-fitting → revert the criteria; root-cause the AI agent.
- Organizational → process retro + the specific mitigations (§4).

Step 5: Prevent recurrence

- Strengthen detection (a lower threshold, a new metric).
- Add a mitigation to the processed artifact.
- Record lessons learned in the project decision log or in the ADAPT backward findings (category `scope` / `terminology`).

Step 6: Verify

After the fix — re-run QG-2 on the affected artifacts. Drift detection should show a clean state.

---

6. Negative: what this chapter does not cover

- **Security incidents** — breach response, forensics, regulatory notification. This is an organization-level security process, not RENAR scope.
- **AI red team / penetration testing** — a separate security workflow; RENAR only tracks that the corresponding SR / SPEC-SEC should exist.
- **Compliance breach response** — a violation of GDPR / FZ-152 / PCI-DSS requires a legal process with the DPO / regulator, not a technical recovery.
- **Production incidents** — outages, performance regressions. These are operational; see the SPEC-OPS runbook.
- **Stakeholder conflicts** — disputes at acceptance, scope disagreements. RENAR provides the audit trail (who approved what, when), but resolution is a human process.

---

7. Relationship to other materials on failure modes

Document	What is in it	When to read
reference/03-ai-risk-register	The full register of 14 AIR risks with mitigations	When planning an AI use case; when reviewing the eval strategy
standard/04-terms §4.11	The closed list of drift classes with normative definitions	When disputing the terminology of failure modes
05-safe-comparison §9	The RACI matrix — who is accountable for each activity	When investigating an organizational failure
reference/04-ai-style-guide	The style of AI provenance; the minimal contract for AI-generated artifacts	When diagnosing AIR-01 (hallucination), AIR-07 (citations)

---

8. Resolved decisions for v1.0

- **A set of recovery steps with no platform binding.** The sequence in §5 is universal in nature. The details of "how exactly to freeze changes" for `git` and a document store are in 03-tool-guide-git §3 and 04-document-store-substrate. The scope of the normative minimum is set right here, in chapter 7.
- **Tuning the critic event-driven.** Re-tuning the critic's prompt is performed when the drift / hallucination metrics breach their threshold (§12.3.3); the RENAR-5 level requires continuous evaluation (§11.8.1), so a regular "general review for no reason" is redundant. On a metric trigger — it is permitted.

8.1 Deferred to v1.1 (phase-8 backlog)

- **Numeric thresholds for the organizational patterns (§4).** Today only qualitative "signs" are given. A set of acceptable values will be needed once field data has accumulated. Owners: the RENAR standard team and the adopting organizations.
- **A formal measurement of the "bus factor" for §4.7.** The supporting tooling is not fixed; a possible approach is a graph query over commit authors across the revision chain (a built-in **V6** combination at the substrate). Owners: the authors of tooling for specific storage environments.

---

08. Developer Guide

> ***Example for the `git` substrate only.*** *This chapter illustrates **one** possible scenario, using GitHub and subsystems split across repositories (`.req` and `.src` directories).* ***RENAR is not tied to any specific substrate implementation****; your hooks and pipelines may be arranged differently. The generally applicable chapters are §6 Requirements hierarchy, §8 Specifications, §10 Lifecycle and quality gates. For an alternative that does not use a VCS as the file base, see 04-document-store-substrate.md.*
>
> ***Fictional company AcmeCorp:*** *the `acme-platform` platform and four subsystems — `acme-portal`, `acme-ai`, `acme-orchestrator`, `acme-site`. Replace the names in the examples with your own; references to the normative chapters are unaffected.*

1. What you need to know before you start

Two repository types. Each subsystem has two separate repositories:

Repository	Suffix	What's inside	Who writes it
Requirements	`.req`	BR, SR — what the system must do	Architect, Tech Lead
Source code	`.src`	Code, tests, CI/CD	Developer

The split is deliberate: requirements are versioned independently of the code, have a different review cycle, and have different access rights.

Hierarchy: System (`acme-platform`) → Subsystem (`acme-portal`, `acme-ai`, `acme-orchestrator`, `acme-site`) → Module (if present). Each level has its own `.req` repository; upper-level requirements are parents for the requirements below them.

Three requirement levels:

```
BR  — Business Requirement  (why the business needs it)
 └── SR  — System Requirement  (what the system does)
      └── TR  — Task Requirement  (the specifics for implementation)
                ↑ these are the fields of your task in the tracker, not a file
```

TR is not a file — it is Goal + Acceptance Criteria in a tracker task.

2. Initial setup of the local environment

Step 1. Confirm with your Tech Lead which subsystem you are working on: `acme-portal` (the customer portal), `acme-ai` (the AI pipeline), `acme-orchestrator` (the orchestrator), `acme-site` (the public site).

Steps 2-3. Create the structure and clone the repositories:

```
mkdir -p ~/projects/acmecorp/acme-platform && cd ~/projects/acmecorp/acme-
platform

# Required for everyone — the system-level parent requirements (read-only):
git clone git@github.com:acmecorp/acme-platform/acme-platform.req

# Required — your subsystem's repositories:
SUBSYSTEM=acme-portal   # replace with your own
mkdir -p $SUBSYSTEM
git clone git@github.com:acmecorp/acme-platform/$SUBSYSTEM/$SUBSYSTEM.req
$SUBSYSTEM/$SUBSYSTEM.req
git clone git@github.com:acmecorp/acme-platform/$SUBSYSTEM/$SUBSYSTEM.src
$SUBSYSTEM/$SUBSYSTEM.src

# As needed — the requirements of adjacent subsystems (read-only):
mkdir -p acme-ai && git clone git@github.com:acmecorp/acme-platform/acme-ai/acme-
ai.req acme-ai/acme-ai.req
```

Step 4. Verify: `find ~/projects/acmecorp -maxdepth 4 -name ".git" -type d | sed 's|/.git||' | sort`. The expected result for an Acme Portal developer:

```
~/projects/acmecorp/acme-platform/acme-platform.req
~/projects/acmecorp/acme-platform/acme-portal/acme-portal.req
~/projects/acmecorp/acme-platform/acme-portal/acme-portal.src
```

3. Folder structure on the local machine

The local structure **mirrors** the repository hierarchy in the hosting platform — the relative `../..` paths between repositories become predictable.

> *For the directory layout (the substrate layout) see guide/03 §2 and §4: the canonical repository scheme + pinning `.src` → `.req` via a submodule (capability V5). Here is the typical local placement for an IDE: several clones lying side by side, without a mandatory submodule in the working folder.*

```
~/projects/acmecorp/acme-platform/
  acme-platform.req/          # system requirements (read-only)
  acme-portal/
    acme-portal.req/          # your subsystem's requirements
    acme-portal.src/          # your subsystem's code — this is where you work
  acme-ai/{acme-ai.req,acme-ai.src}/             # cloned as needed
  acme-orchestrator/{acme-orchestrator.req,acme-orchestrator.src}/
  acme-site/{acme-site.req,acme-site.src}/
```

> ***Rule:** do not change the folder names — they match the project names in the git hosting platform. This lets scripts and CI work with the same paths everywhere.*

4. Setting up the workspace in the IDE

VS Code Workspace. Create `<subsystem>.code-workspace` in the subsystem folder:

```
cat > ~/projects/acmecorp/acme-platform/acme-portal/acme-portal.code-workspace <<
'EOF'
{
  "folders": [
    { "path": "acme-portal.src", "name": "Code — Acme Portal" },
    { "path": "acme-portal.req", "name": "Requirements — Acme Portal" },
    { "path": "../acme-platform.req", "name": "Requirements — System (read only)"
}
  ],
  "settings": { "files.exclude": { "**/.git": true } }
}
EOF
```

Open it: `code ~/projects/acmecorp/acme-platform/acme-portal/acme-portal.code-workspace`. The side panel shows all three repositories at once.

JetBrains (IntelliJ, WebStorm, PyCharm). Open each repository as a separate module via **File** → **Attach Project** or **Project Structure** → **Modules**.

5. Working with requirements

5.1 How to read. Before starting a task, read the SR (`acme-portal.req/sr/SR-NN-*.md`). In the SR frontmatter, find the `parent` field — a link to the parent BR:

```
parent: { id: BR-01, repo: "acmecorp/acme-platform/acme-platform.req", file:
"br/BR-01-order-ai-dev.md" }
```

Open the parent BR — this is the "why the functionality exists".

5.2 Tracing. The chain: `Task TASK-42 → SR-01 (acme-portal.req/sr/...) → BR-01 (acme-platform.req/br/...)`. If anything is unclear, move up the chain.

5.3 Changing a requirement. The developer opens an Issue in the `.req` repository describing the mismatch between the SR and the actual behavior. The Tech Lead / Architect makes the change:

```
cd acme-portal.req
git checkout -b change/TZ-2026-002-totp
# Edit the SR file; update version + updated in the frontmatter; update
source.tz-section to the delta-TZ section
git add sr/SR-01-auth.md
```

```
git commit -m "[delta:TZ-2026-002] SR-01 v1.2: add TOTP"
# Open an MR/PR in the git hosting platform
```

> ***Forbidden:*** *committing requirement changes directly to* `main` *without an MR/PR and review.*

5.4 What you must not do. Change `acme-platform.req` without the Architect's agreement; change the requirements of adjacent subsystems (`acme-ai.req` , `acme-orchestrator.req`); delete requirement files (only move them to `status: deprecated`); create version files (`SR-01-v1.md` , `SR-01-v2.md`) — the history lives in `git log` .

---

6. Working with tasks

6.1 Before you take a task — all QG-0 Approval Gate items are satisfied (reference/01 §2.4; pre-v1.0 legacy "Context Gate"):

- The Goal is formulated; Acceptance Criteria exist — concrete, testable, independent.
- There is at least one negative scenario in the AC.
- The task references an SR (a field in the tracker); a work type is assigned.
- If it touches security — `threat-surface` is declared.

If something is missing — **do not take the task into work**; go back to the Supervisor/Tech Lead.

6.2 Acceptance Criteria. Each AC is a separate test. Good AC: `POST /auth/login returns 200 and a JWT token with valid credentials` ; `POST /auth/login returns 401 with an incorrect password` (a negative scenario); `POST /auth/login returns 422 if the email field is missing` ; `the JWT token expires after 24 hours` .

Bad AC (do not take such a task): "Login should work correctly"; "Error handling should be robust"; "The system should be secure".

6.3 If an AC is unclear or contradicts the SR: read the SR + the parent BR; add a comment to the task with a concrete question; do not interpret silently — an AI interprets silently, which is exactly why clear requirements are needed.

---

7. The Git process

7.1 Working with code (`.src`). The standard feature-branch workflow:

```
cd acme-portal/acme-portal.src
git checkout main && git pull
git checkout -b feat/TASK-42-totp-auth
# ... work ...
git add src/auth/totp.py
git commit -m "feat(auth): implement TOTP authentication (TASK-42)"
# Open an MR/PR in the git hosting platform
```

Commit format: `<type>(<scope>): <description> (<TASK-ID>)` .

7.2 Working with requirements (`.req`). The branch for changing a requirement (see §5.3 above).

7.3 Linking the MR/PR to the task. In the MR/PR description, specify: `Closes #42` + `Related SR: SR-01 (acme-portal.req)` . If the change touches several `.req` repositories — `Related: acmecorp/acme-platform/acme-platform.req#15` .

7.4 Branch naming:

What you're doing	Branch
New functionality	`feat/TASK-NN-slug`
Bug fix	`fix/TASK-NN-slug`
Changing a requirement	`change/TZ-YYYY-NNN-slug`
New requirement	`feat/BR-NN-slug` or `feat/SR-NN-slug`

8. Common scenarios

Scenario 1. A new task: open it in the tracker → check QG-0 → find the link to the SR → open the SR in `acme-portal.req/sr/` → read the SR + the parent BR (if context is needed) → run the TCs bound to the task (they are already frozen and **red** — that is normal: they were written before you and not by you, standard/09 §9.18) → create the `feat/TASK-NN-slug` branch in `.src` → implement until the TCs turn green → MR/PR → review → merge.

> ***Why the test is already written, and not written by you.*** *A test born next to the code checks the code, not the requirement. So the TC is frozen before the task starts and is written by a different agent or engineer. If while implementing you added an internal technical detail that is not in the requirements (a defensive check, logging, an edge-case branch), it can be legalized through the class* `source: implementation-originated` *(standard/06 §6.13): it needs provenance, explicit approval by a human supervisor, a covering test before the merge, and a* ***killed mutant*** *(such a test has no red history — it is born green). Client-observable behavior is not legalized this way: a screen, a field, a message, a deadline — those go through the contractual loop only.*

Scenario 2. A mismatch between the SR and the task's requirement: do NOT do anything silently → add a comment to the task ("SR-01 §4 describes X, the task requires Y — a contradiction, please clarify") → wait for the Supervisor/Architect's answer → do not take the task into work until it is resolved.

Scenario 3. Understanding where a requirement came from. In the `.req` repository: `git log -- sr/SR-01-auth.md` (the change history) or `git show <commit-hash>` (the details of a specific change). Or, in the file's frontmatter, find the `source` field — a link to the TZ and its section.

Scenario 4. Integration (Acme Portal → Acme AI). Clone the requirements of the adjacent subsystem and add them to the workspace; open `acme-platform.req/specs/int/SPEC-INT-01-acme-portal-acme-ai.md` — the contract is described there. Integration contracts in canonical form are `SPEC-INT-NN` (standard/08 §8.5.4), not the legacy `INT-SR` . The subsystem's SR references them via `constrained-by: [SPEC-INT-NN]` . A SPEC-INT always lives in `acme-platform.req/specs/int/` — never inside the subsystems.

Scenario 5. Delta-TZ. This is the work of the Architect/Tech Lead, but for the developer: wait for the MR with the updated SRs to merge → `git pull` in `.req` → read `tz/TZ-YYYY-NNN-delta-N.md` (the delta-TZ text) → check whether the changes affect your current tasks → if so, update or close them in agreement with the Supervisor.

Scenario 6. Updating the local requirements repositories:

```
find ~/projects/acmecorp -name "*.req" -type d | while read repo; do
  echo "Updating $repo..."
  git -C "$repo" pull --ff-only
done
```

9. Access rights — who can change what

Repository	Developer	Tech Lead	Architect
`acme-platform.req` (system BR/SR)	Read-only	Read-only	Write via MR
`acme-portal.req` (subsystem SR)	Read-only	Write via MR	Write via MR
`acme-portal.src` (code)	Write via MR	Write + review	—
`acme-ai.req` (another subsystem)	Read-only	Read-only	—

If you need to change a requirement — open an Issue in the `.req` repository; do not commit directly.

10. Quick reference

Where to look when something is unclear:

Question	Where
What should the system do?	`acme-portal.req/sr/SR-NN-*.md`
Why does the business need it?	`acme-platform.req/br/BR-NN-*.md`
Where did this requirement come from?	frontmatter `source` → TZ
How did the requirement change?	`git log -- sr/SR-NN-*.md`
How does the integration with Acme AI work?	`acme-platform.req/specs/int/SPEC-INT-01-*.md`
What changed with the latest TZ?	`acme-platform.req/tz/TZ-YYYY-NNN-delta-N.md`

Checklist before creating an MR: the code covers all the ACs from the task; there are tests for the negative scenarios; the MR/PR contains a `Closes #NN` reference; if behavior changed — an Issue was opened in `.req` to update the SR.

Git commands (the most frequent):

```
git -C <portal.req> pull                                    # update the requirements
git -C <portal.req> log -- sr/SR-01-auth.md                 # the history of a
specific SR
grep -r "id: SR-01" ~/projects/acmecorp/ --include="*.md"   # find a requirement
by ID
grep -r "SR-01" ~/projects/acmecorp/ --include="*.md"       # all tasks
referencing SR-01
```

Glossary: BR — Business Requirement; SR — System Requirement; TR — Goal + AC in the tracker; AC — Acceptance Criteria; QG-0 — Approval Gate (canonical v1.0; pre-v1.0 legacy "Context Gate"); `.req` — the git repository with the subsystem's requirements; `.src` — the git repository with the code; SPEC-INT — an integration specification (canonical v1.0; pre-v1.0 `INT-SR`); delta-TZ — an addition to the TZ on a repeat order; deprecated — an outdated requirement (not deleted, only flagged).

09. Examples library

> *Seven scenarios (E1–E7): five in-doc examples (E3–E7) plus two walkthroughs (E1–E2). Each in-doc example follows the chain **TZ → ADAPT → BR/SR → SPEC → TC → QG** in full or in part.*

#	Scenario	Document	Audience	Focus
E1	Email/password sign-up (minimal)	00-quickstart	Beginner	30 min, minimal artifacts
E2	Login + profile + permissions (full)	01-walkthrough	Tech Lead, QA	Full lifecycle, AI generation of TC
E3	Personal-data export (GDPR / FZ-152)	§2	Legal, PM, Architect	Compliance, SPEC-DATA/SEC
E4	Webhook idempotency (REST API)	§3	Backend, Architect	`tc-type: contract`, SPEC-API
E5	RAG assistant (SPEC-AI eval)	§4	AI engineer	`tc-type: eval`, judge isolation
E6	Delta-TZ: scope dispute	§5	PM, Client, Architect	ADAPT backward, immutable TZ
E7	A subsystem as a standalone product	§6	Architect	The `implements[]` edge, §6.8.2

2. E3 — Personal-data export (GDPR Art. 15 / FZ-152)

Context: the SaaS "AcmeCRM" stores customer PII. The regulator and the contract require a machine-readable export to be delivered on a data-subject request within ≤30 days.

2.1 TZ fragment (immutable)

```
# TZ-2026-002 — Data subject export

§3.1 A data subject can request a full export of their PII via the UI "Privacy → Export my data".
§4.2 Format: JSON + CSV bundle, zip, SHA-256 checksum.
§4.3 SLA: the download link is ready within ≤ 72 hours of verified identity.
§4.4 The export includes: profile, activity log for 24 months, marketing consents.
§4.5 The request is logged; a repeat export — no more than once every 30 days without an administrator override.
```

2.2 ADAPT (abridged)

```
id: ADAPT-002
type: ADAPT
status: approved
source-tz: { id: TZ-2026-002, signed-date: "2026-05-01" }
```

Forward §3: the export is asynchronous (job queue); identity — via the existing MFA flow.

Backward (resolved):

#	Finding	Resolution
B-01	"24 months of activity" — calendar or rolling?	Rolling 730 days
B-02	Admin override — who approves?	Role `privacy-officer` + an audit log

2.3 BR + SR

```
# br/BR-02-data-subject-export.md
id: BR-02
type: BR
title: "The data subject receives a machine-readable PII export"
status: approved
source: { adapt: ADAPT-002, adapt-section: "Forward §3" }
compliance:
  - { standard: GDPR, control: "Art. 15" }
  - { standard: "FZ-152", control: "art. 14" }
```

```
# sr/SR-08-export-request.md
id: SR-08
type: SR
parent: { id: BR-02 }
title: "An authenticated user initiates an export job"
status: approved
constrained-by: ["SPEC-API-04", "SPEC-DATA-02", "SPEC-SEC-03"]
```

```
# sr/SR-09-export-delivery.md
id: SR-09
type: SR
parent: { id: BR-02 }
title: "The user downloads the ready bundle via a time-limited signed URL"
status: approved
constrained-by: ["SPEC-API-04", "SPEC-SEC-03"]
```

2.4 SPEC (excerpts)

SPEC-DATA-02 — the export-bundle schema (tables: `users`, `activity_events`, `marketing_consents`).

SPEC-SEC-03 — signed URL TTL 24h; rate limit 1 export / 30 days; role `privacy-officer` for the override.

SPEC-API-04 — `POST /v1/privacy/export`, `GET /v1/privacy/export/{job_id}`.

2.5 TC (pos/neg for SR-08)

```
---
id: TC-080
title: "An export job is created for a verified user"
type: TC
tc-type: system
status: ready
verifies:
  - id: SR-08
    requirement-version: "1.0"
negative: false
---
## When
POST /v1/privacy/export (session: verified-user)
## Then
- 202 + job_id; status pending; audit event recorded
```

```
---
id: TC-081
title: "Export rejected without an MFA-verified session"
type: TC
tc-type: security
status: ready
verifies:
  - id: SR-08
    requirement-version: "1.0"
negative: true
---
## When
POST /v1/privacy/export (session: password-only, no MFA)
## Then
- 403; job not created; security audit event
```

Analogous pairs for SR-09 (signed URL valid / expired).

2.6 Quality gates

Quality gate	Evidence
QG-ADAPT-approve (by meaning, near the "architecture gate": `QG-3`)	ADAPT-002 in `approved`, backward findings closed
QG-2 Verification Gate	`TC-080` … `081` pass; the `requirement-version` in `last-run` matches (`1.0`)

2.7 What next

- The full scenario under `git` — 03-tool-guide-git
- Compliance mapping — 06-compliance
- Conformance manifest — reference/08

---

3. E4 — Webhook idempotency (SPEC-API)

Context: a payment provider sends `POST /webhooks/payment` with an `Idempotency-Key`. Duplicates MUST NOT create a double charge.

TZ (fragment): "A repeat webhook with the same key within 24h returns the same `payment_id`, HTTP 200."

SR + SPEC:

```yaml
id: SR-20
type: SR
constrained-by: ["SPEC-API-07"]
```

SPEC-API-07 — the contract: headers, body schema, response codes 200/400/409.

TC (contract pair):

```yaml
id: TC-200
type: TC
tc-type: contract
verifies: [{ id: SR-20, requirement-version: "1.0" }]
negative: false
```

```yaml
id: TC-201
type: TC
tc-type: contract
verifies: [{ id: SR-20, requirement-version: "1.0" }]
negative: true
```

Neg: a second key with a different body → 409 Conflict.

Quality gate `QG-2` : both `TC` s pass; the `requirement-version` in `last-run` matches (`1.0`).

4. E5 — RAG assistant (SPEC-AI)

Context: an in-app chat answers from the knowledge base; eval against a golden Q&A set.

SPEC-AI-02 — model card, fallback, cost cap.

TC eval (judge ≠ production):

```yaml
id: TC-300
type: TC
tc-type: eval
verifies: [{ id: SPEC-AI-02, requirement-version: "1.0" }]
judge:
  vendor: "<vendor>"
  model: "eval-judge-v2"          # ≠ the production model of SPEC-AI-02 (§9.6.2)
baseline:
  artifact: "<golden Q&A dataset>"
  metric-thresholds: {}
negative: false
```

Neg eval: prompt injection in the user message → refusal + audit event (`tc-type: eval` , adversarial negative).

See standard/09 §9.6.2, guide/07 §3.5.

5. E6 — Delta-TZ: scope dispute

Context: after the signed TZ the client asks to "add PDF export" — outside the scope of ADAPT-002.

The correct path to the Source of Truth (SoT):

1. Do **not** edit the immutable TZ and do **not** silently extend the SR.
2. Draw up a **delta-TZ**; the adversarial reviewer reviews it and issues a verdict (§7.6.1). On a "findings present" verdict a delta-ADAPT `ADAPT-002-delta-1` is created (forward + backward); on "no findings" no delta-ADAPT is created and derived artifacts reference the delta-TZ via `source.tz-section` .
3. Backward finding: "PDF export was not part of Forward §3 of ADAPT-002."
4. The client's decision on the finding is drawn up as an **ACTZ** — a TZ clarification protocol with a dual signature (standard/07 §7.13); the finding receives `decided-in: ACTZ-003 §1` .
5. The delta-ADAPT is approved by the Architect's signature (§7.5) → new SR/SPEC/TC. The effective TZ is recomputed, and the ATs are regenerated from the new revision.

Anti-pattern: a direct commit to `sr/SR-08` without a `delta-ref` — drift class 4.11.6 (standard/04 §4.11).

See standard/07 §7.6, guide/02-transition-guide.

6. E7 — A subsystem as a standalone product (`implements` -edge)

Context: the platform `acme` (a system) consists of the subsystem `acme.notify` — a separate product with its own business owner (Notify Lead), its own team, and its own release cycle. Scenario §6.8.2 of the RENAR Standard.

6.1 Artifact hierarchy

```
acme (system)
├── BR-01 (order intake with AI assistance)                level: system
├── BR-05 (monitoring and alerts for the operations team)  level: system
└── acme.notify (subsystem, standalone product)
    └── BR-01 (multichannel notification delivery)         level: subsystem
        implements:
          - id: BR-01      (elaborates "order intake": notifications on status
changes)
            scope: { system: acme }
          - id: BR-05      (elaborates "monitoring": notifications on SLA
breaches)
            scope: { system: acme }
        ↓
        SR-01..SR-12 (notify-internal requirements)
        SPEC-INT-01 (integration with acme via the message bus)
        SPEC-API-01 (REST API for notify clients)
```

`acme.notify.BR-01` is the root node of its own requirements tree (`parent` is absent, as for any BR). The link to the system is expressed by a **typed** `implements[]` edge, not a parent-edge.

6.2 frontmatter `acme.notify/br/BR-01-multichannel-delivery.md` (fragment)

```yaml
---
id: BR-01
title: "Multichannel notification delivery"
type: BR
level: subsystem
scope:
  system: acme
  subsystem: acme.notify

status: approved
owner: "Notify Lead (notify-side); Architect (acme-side)"

source:
  adapt: ADAPT-NOTIFY-001
  adapt-section: "Forward §2"
  tz-section: "TZ-NOTIFY §3"
```

```yaml
# === implements-edge §6.8.2 ===
implements:
  - id: BR-01
    scope:
      system: acme
    rationale: "ADAPT-NOTIFY-001 §2.1 — notifications on order status change"
  - id: BR-05
    scope:
      system: acme
    rationale: "ADAPT-NOTIFY-001 §2.4 — SLA alerts for operations"

business-context:
  stakeholder: "Notify Lead"
  business-goal: "Timely delivery of notifications to the end-customer and the
operations team"
---

# BR-01: Multichannel notification delivery

The notify subsystem delivers notifications …
```

6.3 Machine-readable trace chain

```
TC-NOTIFY-15
  → verifies SR-NOTIFY-08 (rate-limit for the email channel)
        ├─ parent:   acme.notify.BR-01 v1.2
        │              └─ implements: acme.BR-01 v3.0, acme.BR-05 v1.4
        │                                (typed cross-level edge)
        ├─ source.adapt: ADAPT-NOTIFY-001 §Forward §2.2
        └─ constrained-by: SPEC-API-01, SPEC-OPS-01   (scope: acme.notify)
```

On an audit, the full chain is reconstructed from TC-NOTIFY-15: SR → subsystem BR → system BR. Before v1.0 (without the `implements`-edge) the chain broke off at `acme.notify.BR-01` and continued only through the prose of the "Context" section.

6.4 Gates on the substrate side

When `acme.notify.BR-01` is approved, the substrate-native hook checks (see standard/10 §10.11.1):

1. `acme.BR-01` and `acme.BR-05` exist in the substrate by `id + scope.system`.
2. Both target BRs are in status `approved` (or `verified`).
3. The chain `acme.notify.BR-01 → acme.BR-01 → …` does not form a cycle.
4. `acme.notify.BR-01.level = subsystem` (rule: `implements[]` does not apply at `level: system`).

If any check fails, the approval is blocked. Behavior when `acme.BR-01 = deprecated`: a warning, not fatal (the Architect decides: update `implements` to the current BR or mark `acme.notify.BR-01` as requiring review).

6.5 Evolution "module → subsystem" via `implements`

When a module acquires a business owner (standard/06 §6.9.1):

1. The business owner is captured via an ADAPT backward finding (category `scope`).
2. After the delta-ADAPT is approved, the module is promoted to a subsystem; a subsystem BR is created.
3. **New step (v1.0+):** the subsystem BR declares `implements[]` on the applicable system BRs. Without this step, the §6.8.2 scenario carries an absence of traceability incompatible with v1.1 conformance.

Anti-pattern: creating `acme.notify.BR-01` with `level: subsystem` but without `implements[]` — formally permitted on v1.0 (recommended), but non-conformant on v1.1+. If the parent system has an approved BR, the absence of `implements[]` MUST be **explicitly justified** in the "Context" section of the BR with a reference to ADAPT§.

See standard/06 §6.5.2, §6.8.2, §6.10.3, standard/13 §13.3.8.

---

10. Migration to v1.0

> *For teams that already managed requirements "their own way" or on early RENAR drafts. The goal is **no big-bang**: preserve immutable IDs, replace deprecated constructs, and issue a new conformance manifest. The normative basis — standard/04 §4.14, CHANGELOG §Migration.*

Do not confuse this with 02-transition-guide — that covers a staged entry into RENAR-1..5 from scratch; here we deal with a **breaking rename** and schema alignment when moving to the current edition of the standard.

1. When this migration is needed

Situation	Action
A project with no RENAR, but with a TZ + Jira	First 02-transition-guide, not this document
Files with `INT-SR`, `INT-TC`, `AIC`, `UIC`, `TS`	This guide
`verifies[].version` without `requirement-version`	Update the TC frontmatter (reference/02 §8)
Manifest `renar-version: 0.1-draft` or absent	Issue a new manifest (reference/08)

2. Type-replacement table (closed list, v1.0)

Deprecated	Canonical	Migration step
`INT-SR`	`SR` + `constrained-by: [SPEC-INT-N]`	Rename the type; create / bind a SPEC-INT
`INT-TC`	`TC` + `tc-type: contract`	Add `type: TC`, `tc-type: contract`
`AIC`	`SPEC-AI`	Move the body into the SPEC-AI frontmatter
`UIC`	`SPEC-UI`	Move baselines into `specs/ui/baselines/`
`TS`	`SPEC-ARCH` or `SPEC-OPS`	By content (architecture vs ops runbook)
`TM` (module SR type)	`SR` + `level: module`	Drop the pseudo-type; set the level
`tc-type: acceptance`	`tc-type: business`	Rename the value. There are now two acceptances: a `business` TC measures the BR's business goal (the internal loop), while conformance to the contract is checked by `AT` (standard/09 §9.19) — one name for two different things has been removed (§9.5)

`approval.client-signature` on an ADAPT	The Architect's signature alone	Remove the client signature from the ADAPT; move the client's decisions into signed `ACTZ` s and set `decided-in` on the findings (standard/07 §7.5, §7.13)
ADAPT state `client-ready`	`asked`	Removed from the state machine: putting questions to the client is the business of the ACTZ, not a state of the ADAPT (standard/10 §10.8)

Do not change IDs. If a filename contains a legacy prefix, a `legacy-id` field in the frontmatter is acceptable (informative traceability).

3. Step-by-step plan (1–2 sprints)

Phase A — inventory (1–2 days)

1. `grep` / search across the substrate: `INT-SR` , `INT-TC` , `AIC` , `UIC` , `type: system` (the erroneous TC type).
2. List the artifacts with broken `verifies` / missing `source.adapt` .
3. Record the current `RENAR-CONFORMANCE.yaml` (if any) as the baseline state.

Phase B — schema pass (3–5 days)

1. Batch-rename the types per the §2 table (one PR per type, or one atomic change-set per delta-TZ).
2. TC: `type: TC` , `tc-type` , `verifies[].requirement-version` , `last-run.requirement-version` .
3. SR: `constrained-by[]` on every SPEC in use.
4. BR: the adaptation source — `source.adapt` on the approved ADAPT, or `source.adversarial-review-ref` on the issued AR when the verdict is "no findings"; `source.tz-section` is always mandatory (standard/07 §7.4.1).

Phase C — validation (1–2 days)

1. Substrate hooks / CI: the frontmatter validator (reference/02).
2. Run the TCs; update `last-run` .
3. Self-assessment checklist — reference/08 §2.

Phase D — manifest (1 day)

1. Bump `renar-version: "1.0"` .
2. Increment `manifest-version` ; new `manifest-id` .
3. Signature of the Architect / Tech Lead (V6).

4. Common pitfalls

Mistake	Consequence	Fix
Renaming `SR-05` → `SR-05-v2`	V1 immutable-ID violation	`deprecated` + a new ID with `replaces`
Leaving `type: system` on a TC	Validator fail; KG drift	`type: TC` + `tc-type: system`
Migrating code before the `.req`	SoT inversion violated	First the SR/SPEC/TC approved, then the TR
Skipping the adversarial review "only for new TZs"	Non-conformant for legacy TZ	A retrospective adversarial review of every active TZ; an ADAPT only on a "findings present" verdict (standard/07 §7.4.1)

5. Rollback

The migration runs through the substrate history (V1). Rollback means reverting the change-set / PR, **not** deleting artifacts. Deprecated artifacts remain with `status: deprecated` for audit.

6. Related documents

Document	Why
02-transition-guide	RENAR-1..5 without schema breaking changes
09-worked-examples	Reference frontmatter after migration
reference/07	External ISO 29148 statement
standard/13 §13.7	Re-assessment after migration

RENAR Guide 1.0 — renar.tech

RENAR Glossary

> ***Purpose:*** *the single source of canonical RENAR terms with examples and a mapping to industry standards. On a wording conflict,* `standard/04-terms.md` *wins* ***normatively****; this glossary is an informative lookup for the reader and the assessor.*

1. Authority chain

When a term is disputed, the following order applies:

1. `standard/04-terms.md` — the normative canon: a standard chapter's definitions win on any conflict.
2. **This document** — informative clarifications and mapping to industry standards; does not override `standard/04`.
3. **ISO/IEC/IEEE 29148** — the international requirements-engineering standard.
4. **BABOK Guide v3** — the *Business Analysis Body of Knowledge*.
5. **A change via an amendment proposal to the full RENAR Standard** — when all sources are silent.

Closed lists: the master index of the closed lists is `standard/01 §1.7.5`.

Not used as a source of terms: bug-tracker tickets, team chats (slang), outdated slide decks, marketing materials.

2. Canonical terms

2.1 Requirement levels

The v1.0 canon is a closed list (see `standard/04-terms.md §4.3`):

RENAR (canonical)	Full name	RU UI label (reference)	Description
BR	Business Requirement	Бизнес-требование	A customer-level business goal: what the organization wants to obtain.
SR	System Requirement	Системное требование	An engineering requirement for the software; verifiable. One `SR` is one verifiable unit.
TR	Task Requirement	Требование к задаче	An implementation-level requirement derived from an `SR`; the unit of work planning.
TC	Test Case	Контрольный пример (TC)	A verifiable artifact that confirms an `SR` / `BR`. Describes behavior, not implementation.

Identification rule: in `frontmatter`, `id` is the canonical RENAR identifier (`BR-01`, `SR-05`). On export to another substrate, a target mapping applies.

Refinements of `SR` (frontmatter fields, not separate artifact types):

- **Module SR** — an `SR` with `level: module` (`standard/06 §6.7`). Formerly the separate label `TM` (§2.1.1).
- **Integration SR** — an `SR` with `constrained-by: [SPEC-INT-N]`. Formerly the separate label `INT-SR` (§2.1.1).
- **Contract TC** — a `TC` with `tc-type: contract`. Formerly the separate label `INT-TC` (§2.1.1).

The `SPEC` family (UX / AI / architecture / integration specifications) — see §2.5 The `SPEC` family.

2.1.1 Legacy labels (deprecated)

When migrating from pre-v1.0 material, deprecated labels appear. Their replacements in the v1.0 canon (`standard/04-terms.md §4.14.1`):

Legacy label	v1.0 canonical replacement	Note
`TM` (Module/Submodule SR)	`SR` with `level: module`	A refinement of `SR`, not a separate artifact type
`UIC` (UI Concept)	`SPEC-UI` (`standard/08 §8.5.6`)	Part of the `SPEC` family (§2.5)
`AIC` (AI Concept)	`SPEC-AI` (`standard/08 §8.5.7`)	Part of the `SPEC` family
`INT-SR` (Integration SR)	`SR` with `constrained-by: [SPEC-INT-N]`	A subclass of `SR`
`INT-TC` (Integration TC)	`TC` with `tc-type: contract`	A subclass of `TC`
`TS` (Technical Specification)	`SPEC-ARCH` or `SPEC-OPS`, depending on content	Part of the `SPEC` family

Migrating existing artifacts: the substrate-native binding to a change set MUST automatically recognize deprecated labels and offer the canonical replacements. The canonical legacy → v1.0 mapping table is `standard/04-terms.md §4.14.1`.

2.2 ADAPT artifact family

Term	Description
ADAPT	The Adaptive Document for Articulating a Project's TZ. An **internal** artifact of interpretation: the forward direction (construal) and the backward direction (findings). Signed by the **Architect only** (`standard/07 §7.5`) — the client never sees it.
ACTZ	The TZ Clarification Protocol (*Agreed Clarification of TZ*). A **contractual** artifact: decisions put to the client and signed by **both parties**. Its contractual name is "TZ Clarification Protocol No. N". Lifecycle: `draft → sent → signed → superseded` (`standard/07 §7.13`). The boundary with ADAPT is drawn **by audience**: shown to the client and approved → an obligation; not shown → an interpretation.

effective TZ	The baseline for delivery and acceptance: the initial TZ (with its annexes) **plus every signed ACTZ**; on a divergence, the later signed document prevails. AT are derived from the effective TZ (`standard/07 §7.14`). ADAPT is **not** part of the acceptance baseline.
`decided-in`	A field of a backward-finding record: a reference to the clause of a **signed** ACTZ in which the decision is recorded. A finding cannot be `resolved` without it (`standard/07 §7.4.5`).
delta-ADAPT	An `ADAPT` for a delta-TZ. Chain: ADAPT-001 → ADAPT-001-delta-1 → ADAPT-001-delta-2; applied in order.
errata-ADAPT	A correction to an approved `ADAPT` (our own interpretation error). It does not alter the frozen document — a separate artifact is added.
Forward (in ADAPT)	The engineering interpretation of each TZ section: quote → interpretation → elaborated scenarios → coverage.
Backward (in ADAPT)	The list of problems and questions for the client. Lifecycle: `open` → `asked-to-client` → `answered` → `resolved` → `frozen` .
Term mapping (in ADAPT)	A "client term → engineering understanding" table.

2.3 Backward categories (closed list)

Every ADAPT backward entry belongs to one of 7 categories. The list is closed; new ones are added only through a change to the full RENAR Standard:

Category	Description
`contradiction`	A contradiction within the TZ.
`gap`	A gap — the TZ is silent on this.
`hidden-assumption`	A hidden engineering assumption that needs confirmation.
`feasibility`	A technically infeasible or expensive requirement.
`regulatory`	Touches legislation / compliance.
`terminology`	An unclear or conflicting client term.
`scope`	A scope-boundary clarification (in / out).

2.4 Quality Gates (closed list, v1.0 canon)

The closed list of RENAR Quality Gates (per `standard/10-lifecycle-qg.md §10.3–10.4`):

Gate	Applies to	Pass condition
QG-0 Approval Gate	`BR` / `SR` / `SPEC` transition: `draft` → `approved`	Goal, acceptance criteria, at least one negative scenario, and coverage; for `SR` — the parent `BR` in `approved` +; for an `SR` with `constrained-by` — the `SPEC` in `approved` +.

QG-1 Implementation Gate	`TC` transition: `draft` → `ready` (**only** for `TC`)	The `TC`'s `verifies` field points to an artifact in `approved`+; `requirement-version` matches; implementation coverage and the version pin are recorded.
QG-2 Verification Gate	`SR` / `BR` transition: `approved` → `verified`	All `TC`s from `verified-by` are green; at least one `TC` with `negative: true`; `last-run` has a `requirement-version` matching the current `version`; the spot-check passed.
QG-3 Architecture Gate (*optional*)	`SPEC-ARCH` approval / the Architect's signature on an ADAPT	An ADR-style artifact is recorded in the substrate; the Architect's signature. Not REQUIRED for declared RENAR conformance.
QG-4 Acceptance Gate (*optional*)	`BR` transition: `verified` → `accepted`	The BR is in `verified`; the business outcome is measured and has reached the threshold; the Stakeholder's signature. Not REQUIRED for declared RENAR conformance.

RENAR conformance: `QG-0` / `QG-1` / `QG-2` are mandatory for declared conformance (`standard/13 §13.3`). `QG-3` / `QG-4` are optional extensions.

The list is closed; new gate numbers are added only through the formal change procedure of the full RENAR Standard.

2.4.1 Legacy QG names (deprecated)

Before v1.0, RENAR used different gate names. The mapping per `standard/04-terms.md §4.14.1`:

Legacy (pre-v1.0)	v1.0 canonical replacement	Note
`QG-0 Context Gate`	`QG-0 Approval Gate`	Rename only; meaning preserved
`QG-1 Requirements Gate`	`QG-1 Implementation Gate`	**Meaning shift:** formerly `BR` / `SR` approval, now only the `TC` transition `draft` → `ready`
`QG-2 Implementation Gate`	`QG-1 Implementation Gate`	Renumbered and merged into a single `QG-1`
`QG-3 Verification Gate`	`QG-2 Verification Gate`	Renumbered
`QG-4 Acceptance Gate`	`QG-4 Acceptance Gate`	Same name; now **optional** for declared RENAR conformance

Migrating existing artifacts: automation rewrites references per the table above. The canonical legacy QG → v1.0 mapping table is `standard/04-terms.md §4.14.1`.

2.4.2 Local ADAPT gates

The `ADAPT` lifecycle (`standard/07-adapt.md`) uses local `ADAPT` gates alongside the canonical `QG-N`. These are not a separate `QG` numbering but local lifecycle events of the `ADAPT`

artifact:

Gate	Application	Pass condition
QG-ADAPT-draft	`ADAPT` creation	The Forward section covers every TZ section.
QG-ADAPT-review	Transition to `review`	All backward entries are `open` or `asked-to-client`; none are `draft`.
QG-ADAPT-asked	Putting the questions to the client	All backward entries are `asked-to-client`; the questions are put to the client as part of an ACTZ (`status: sent`).
QG-ADAPT-answered	After the client answers	All backward entries are `answered`.
QG-ADAPT-approve	`ADAPT` approval	All backward entries are `resolved` with `decided-in` on a clause of a signed ACTZ; the Architect's signature.
QG-ADAPT-frozen	After approval	Immutability; generation of `BR` / `SR` / `SPEC` is permitted.

Local `ADAPT` gates are **not** part of the `QG-0` / `QG-1` / `QG-2` set for declared RENAR conformance. An implementation MAY express `QG-ADAPT-approve` as a local alias for `QG-3` (`Architecture Gate`, where applicable) or store it separately — at the substrate's discretion.

2.5 The SPEC family (closed list)

Type	Purpose	Source
SPEC-ARCH	System / subsystem architecture: contexts, containers, components, deployment view, quality attributes	ADAPT Forward section
SPEC-API	API contracts (REST / GraphQL / gRPC / async events); versioning, error model, rate limits	ADAPT Forward section
SPEC-DATA	Data model: schema, ER diagram, indexes, migrations, storage, personal-data classification	ADAPT Forward section
SPEC-INT	Integration: interaction between subsystems and external systems; protocols, contracts, SLAs	ADAPT Forward section
SPEC-PROC	Process / workflow: business processes, state machines, saga patterns, orchestration and choreography	ADAPT Forward section
SPEC-UI	UI / UX: screens, navigation, user scenarios, accessibility, localization, baseline images	ADAPT Forward section
SPEC-AI	AI / ML: model cards, RAG, prompt engineering, evaluation strategy, cost budget	ADAPT Forward section
SPEC-SEC	Security: authentication / authorization, threat model, secrets management, data classification	ADAPT Forward section, backward regulatory findings

SPEC-OPS	Operations: deployment, observability, SLO / SLA, runbooks, disaster recovery	ADAPT Forward section
SPEC-TEST	Test benches and data: topology, emulators and sandbox instances of external systems, run configuration, datasets on both sides of every integration, reconciliation rules, volume and anonymization; a client signature when real data is used	ADAPT Forward section, integration requirements
SPEC-DOC	Delivered documentation: composition (user manual, administrator manual, training materials), the mandatory sections of each document, version binding, acceptance criteria	ADAPT Forward section, the contractual composition of the delivery

The list is closed — exactly eleven types (canon per `standard/08 §8.3`); new `SPEC` types are added only through a change to the full RENAR Standard.

Three subjects of description are kept apart and MUST NOT be conflated: the nine types (including `SPEC-OPS`) state **how the system is built and how it lives**; `SPEC-TEST` — **how its conformance is proven** (benches, data, run configuration); `SPEC-DOC` — **what enters the delivery** to the client. Hence the boundaries: an administrator manual is always `SPEC-DOC` (the client receives it as part of the delivery), while the vendor team's internal runbook is always `SPEC-OPS`. Every dynamic `TC` is bound to a bench through `environment-ref` pointing at a `SPEC-TEST` (static checks require no bench), and every `SPEC-DOC` is verified by the doc-lint.

2.6 Lifecycle statuses

Status	Applies to	Meaning
`draft`	all artifacts	In progress, not for use by others
`review`	all	Under review, changes possible
`approved`	ADAPT, BR, SR, SPEC	Approved; immutable after signature (for `ADAPT` — the Architect's, §7.5)
`verified`	BR, SR, SPEC	Confirmed by passing `TC` s and the spot-check
`frozen`	ADAPT	Approved and immutable; used as the source for generating `BR` / `SR` / `SPEC`
`deprecated`	all	Outdated, not used in new artifacts; the replacement (if any) is given by the `replaced-by` field
`obsolete`	TR, SPEC, TC	Terminal status: the artifact is no longer current and is not deleted (V1); `ADAPT` has no `obsolete` state
`superseded`	ADAPT, ACTZ, AR	The terminal supersession status: a previously correct decision is revoked by a later artifact; the record is retained for audit (`standard/07 §7.6.4`)

2.7 Substrate capabilities (V1–V6)

RENAR is not tied to a specific artifact substrate. An artifact MAY reside in any substrate that satisfies the following capabilities.

Normative definition — `standard/03 §3.3`:

Capability	Description
V1 — immutable history	Any past state of an artifact can be addressably restored without loss (the revision chain is preserved for audit).
V2 — atomic change unit	A change to an artifact (or a consistent group) commits as a single transaction: fully succeeds or fully rolls back; intermediate states are not externally visible.
V3 — diff & review	A proposed change can be presented as a diff against a base version and accepted or rejected before it reaches the approved state (the basis of approval and the `QG` gates).
V4 — branching & change sets	Work in progress is separated from the approved Source of Truth; several independent changes proceed in parallel without affecting the Source of Truth.
V5 — cross-substrate version pin	A specific version of an artifact in another substrate can be pinned as a resolvable identifier (the basis of the `verifies[].requirement-version` field).
V6 — author + timestamp	For each atomic edit, an unambiguous author and timestamp are recorded (the basis of the `ADAPT` signature and the `ai-provenance` block).

Example substrates: a distributed VCS (`git` with merge-request review), a document-oriented store with a revision chain and signatures, any DBMS with change history and signatures. RENAR does not require `git` — only the **V1–V6** capabilities.

2.8 AI provenance (`ai-provenance` , canonical fields)

In the `frontmatter` of any AI-generated artifact:

Field	Type	Description
`ai-provenance.generated-by`	string	The model, as `<vendor>-<model>-<version>@<date>`. Example: `anthropic-claude-opus-4-7@2026-05-15`.
`ai-provenance.prompt-template`	string	Path to the prompt template and its version. Example: `prompts/adapt-from-tz.md@v2.1`.
`ai-provenance.context-tokens`	integer	Token count of the input context.
`ai-provenance.output-tokens`	integer	Token count of the model output.
`ai-provenance.generation-time-ms`	integer	Generation time in milliseconds.
`ai-provenance.human-edits`	boolean	`true` if a human edited the text after generation. **For approved artifacts this field MUST be `true`.**

2.9 Test-case types

A closed list — six types (canon per `standard/09 §9.5`):

TC type (`tc-type` field)	ISTQB correspondence	Application
`business`	Acceptance Testing	Verifies that the business goal of a BR is achieved (the internal contour). The former name `acceptance` is withdrawn: there are two acceptances, and one name for two subjects caused confusion.
`ux`	usability extension	Verifies a `SPEC-UI` via a VLM judge model / visual comparison against a baseline.
`system`	System Testing	Verifies SR, SPEC-PROC, SPEC-ARCH.
`contract`	Component Integration Testing	Verifies SPEC-API / SPEC-INT / SPEC-DATA via contract testing (Pact and similar).
`eval`	AI-specific	Verifies a `SPEC-AI` via an evaluator LLM and metrics (BLEU, accuracy, hallucination rate).
`security`	security extension	Verifies a `SPEC-SEC` : security invariants (STRIDE); normatively negative scenarios only (`standard/09 §9.6.4`).

2.9bis AT and test-evidence principles

Term	Description
AT (*Acceptance Test*)	The acceptance test of the **contractual contour** (`standard/09 §9.19`). It is derived by an **isolated agent** exclusively from the effective TZ — without access to ADAPT, BR/SR/SPEC, TC, or the code. It is regenerated before every round of trials. It verifies conformance to the **contract**, not to the interpretation.
Test authorship isolation (P8)	The test is frozen before the implementation is started; it is written by an agent **other than** the one writing the implementation; an edit to the criteria is made only through `[test-spec-change]` (`standard/09 §9.18.1`).
Red history (P9)	A test **never once observed red is not evidence**. The fixing run before the implementation MUST be red (`standard/09 §9.18.2`). Isolation guarantees that the test was written before the code, but not that it **verifies anything**; red history closes that remainder.
Killed mutant	The compensation for red history in the `implementation-originated` class: such a test is written after the code and is born green, so its working order is proved by a killed mutant (`standard/06 §6.13.3`).
`implementation-originated`	The narrow legal class of requirements originated by the implementation: **only** internal technical details with no client-observable behavior; with provenance, human approval, a TC before the merge, and a counter in the drift metrics (`standard/06 §6.13`).

2.10 Links (frontmatter fields)

Field	Purpose
`parent`	Parent in the hierarchy (BR → SR → TR); a single source.
`children`	Child artifacts (auto-derived).

`source.adapt`	The `ADAPT` the artifact is derived from (`BR` / `SR` / `SPEC`). The field is **conditional**: present when an ADAPT was created; omitted when the adversarial reviewer returned «no findings» (`standard/07 §7.4.1`).
`source.adapt-section`	The `ADAPT` section (Forward §N).
`source.tz-section`	The TZ section; always mandatory (the dual traceability chain).
`source.adversarial-review-ref`	A reference to an `AR` — the adversarial-review record; mandatory whenever `source.adapt` is omitted (`standard/07 §7.4.6`).
`verifies` (in `TC`)	The `SR` / `BR` the `TC` covers, with `requirement-version` .
`verified-by` (in `SR`)	The `TC` s that confirm the `SR` (auto-derived).
`derived-from`	Template and version (if the artifact was created from a template).
`replaces` / `replaced-by`	Replacement when status is `deprecated` .
`supersedes` (in the new one)	Which requirement is being superseded.
`linked-tasks`	Tasks implementing the `SR` (via the runtime environment, not via files).

2.11 File-naming convention (default)

Type	Pattern	Example
ADAPT	`adapt/ADAPT-NNN[-delta-N].md`	`adapt/ADAPT-001-main.md` , `adapt/ADAPT-001-delta-1.md`
BR	`br/BR-NN-<slug>.md`	`br/BR-01-notification-capture.md`
SR	`sr/SR-NN-<slug>.md`	`sr/SR-05-notification-feed.md`
Subsystem SR	`sr/<MODULE>-SR-NN.N-<slug>.md`	`sr/WMS-SR-01.2-pick.md`
SPEC-UI	`specs/ui/SPEC-UI-NN-<slug>.md`	`specs/ui/SPEC-UI-02-notification-feed.md`
SPEC-AI	`specs/ai/SPEC-AI-NN-<slug>.md`	`specs/ai/SPEC-AI-01-rag-strategy.md`
SPEC-INT	`specs/int/SPEC-INT-NN-<slug>.md`	`specs/int/SPEC-INT-01-auth-billing.md`
SPEC (generic)	`specs/<type>/SPEC-<KIND>-NN-<slug>.md`	`specs/api/SPEC-API-03-orders.md`
TC	`tests/TC-NN-<slug>.md`	`tests/TC-01-login-success.md`
TZ	`tz/TZ-YYYY-NNN.md`	`tz/TZ-2026-001.md`
Delta-TZ	`tz/TZ-YYYY-NNN-delta-N.md`	`tz/TZ-2026-001-delta-1.md`

UX baseline	`specs/ui/baselines/SPEC-UI-NN-<scenario>.png`	`specs/ui/baselines/SPEC-UI-02-feed-default.png`
Eval dataset	`specs/ai/eval-datasets/SPEC-AI-NN-<slug>.jsonl`	`specs/ai/eval-datasets/SPEC-AI-01-typical-queries.jsonl`

This is the default convention; substrate-native stores MAY use a different layout, provided identifiers are stable (capability **V1**).

2.12 Change-record markers (informative)

In a substrate where atomic changes carry metadata (a commit message in `git`, a change-record description in a document-oriented store), the following markers are permitted:

Marker	Purpose
`[delta:TZ-YYYY-NNN]`	A change driven by a delta-TZ.
`[test-spec-change]`	A change to a `TC`'s pass/fail criteria (a separate approval).
`[baseline-update]`	An update to a UX baseline or an eval dataset (a separate approval).
`[coverage]`	Automatic regeneration of coverage, test-plan, and requirement summaries (a bot).
`[reconciliation]`	A change by a reconciliation agent.
`[multi-model-disagreement]`	An artifact where AI model outputs disagree — requires manual review.
`[AI]`	A prefix for AI-generated changes.

A substrate-native mechanism MAY express these markers as change-record fields, labels, or tags — the details are not normative.

3. Mapping to standards

3.1 Requirement levels

RENAR v1.0 canonical labels and external standards:

RENAR	ISO/IEC 29148	BABOK	SAFe	Document store (example enum)	SENAR (RU)
BR	Business Requirement	Business Need	Portfolio Epic / Strategic Theme	`BT`	БТ

SR	System / Software Requirement	Solution Requirement (Functional)	Feature	`ST`	CT
SR (`level: module`)	(subcomponent scope, extension)	(subcomponent scope)	Story (sometimes)	`TM` (legacy)	CT модуля
SR (`constrained-by: SPEC-INT-N`)	Interface requirement	Interface solution	Cross-feature integration	(related)	INT-CT (legacy)
TR	(no direct class; refinement of a system / system-element requirement)	Detailed solution requirement	Story	`TK`	T3
TC	Test Case	Verification	Story acceptance test	`test_case`	TK
TC (`tc-type: contract`)	Interface test	Component Integration Testing	Contract test	(related)	INT-TK (legacy)
SPEC-UI	(between BR/SR — design specification)	Stakeholder requirement (UX fragment)	(design level)	(extension)	UIC (legacy)
SPEC-AI	(REQ extension)	(n/a)	Enabler	(extension)	AIC (legacy)
SPEC-ARCH / SPEC-OPS	Design description	Solution component	Enabler tech spec	(extension)	TC (legacy)

Note: the "document store (example enum)" and "SENAR (RU)" columns contain historical labels (`TM` , `UIC` , `AIC` , `INT-CT` , etc.) for traceability with pre-v1.0 systems. The RENAR canon column holds the v1.0 labels per §2.1.1. On export to a document store or a SENAR substrate, a target mapping applies.

3.2 Quality Gates

A mapping of RENAR v1.0 canonical `QG-N` to external models. Legacy `QG` names (§2.4.1) are retained in the SENAR column for historical traceability.

RENAR (v1.0)	SENAR (legacy mapping)	Document store (example)	CMMI activity
QG-0 Approval Gate	QG-0 (context)	`VK-1` (start)	Requirements review before commitment
QG-1 Implementation Gate	QG-2 (implementation, legacy)	`VK-1`	Implementation baseline (`TC` readiness)

QG-2 Verification Gate	QG-3 (verification, legacy)	`VK-2`	Verification
QG-3 Architecture Gate *(optional)*	(n/a)	`VK-3` (partial)	Architecture-decision approval
QG-4 Acceptance Gate *(optional)*	QG-4 (Acceptance)	`VK-4`	Customer acceptance

Note: before v1.0, `QG-1 Requirements Gate` is effectively split between the canonical `QG-0 Approval Gate` (`BR` / `SR` / `SPEC` approval) and `QG-1 Implementation Gate` (`TC` readiness). See §2.4.1 for the full mapping.

3.3 Lifecycle statuses

RENAR	Document store (example)	ISO/IEC 29148	CMMI
`draft`	`draft`	proposed	identified
`approved`	`approved`	agreed-to / baselined	committed
`verified`	`verified`	verified	validated
`deprecated`	`obsolete`	retired	obsolete

3.4 Process artifacts

RENAR	BABOK	SAFe	SENAR
ADAPT (Forward + Backward)	Requirements Analysis Document	Solution Intent (fixed + variable)	(n/a — RENAR extension)
Work Order / TZ	Stakeholder commitment artifact	Customer order	(context)
delta-TZ	Change Request	(n/a — handled via Solution Intent updates)	(context)
Impact Analysis	Impact Analysis (BABOK §8)	(derived)	(context)
Spot-check	Random sampling QA	(n/a)	Rule 9.5
Adversarial review	Independent verification	(n/a — REQ extension)	(via ADR metric)
Reconciliation	Continuous improvement audit	Inspect & Adapt	Quality Sweep

3.5 User-interface projections

`frontmatter` fields, identifiers, and file names are always canonical (latin). RU labels are permitted in the UI:

Canonical	UI (RU)
Business Requirement	Бизнес-требование
System Requirement	Системное требование
Test Case	Контрольный пример (`TC`)
Quality Gate	Контрольная точка качества
Acceptance	Приёмка
Verified	Проверено
Approved	Утверждено
Deprecated	Устарело
Frozen	Замороженный
Backward finding	Замечание к ТЗ
Forward interpretation	Инженерная интерпретация

A UI projection does not replace the canonical identifiers in the substrate.

4. Forbidden / deprecated terms

RENAR does not use the following terms (even where they appear in SENAR / industry literature):

Term	Use instead	Why
User Story as a requirement	SR	A story is a unit of planning, not a requirement. A story MAY implement an SR, but is not itself a requirement.
Use Case (formally)	SPEC-UI + SR	A use case mixes UX and behavior. RENAR separates SPEC-UI (the UX baseline) and SR (behavior).
Spec (without a qualifier)	SR / BR / SPEC-API / SPEC-DATA / ...	"Spec" is ambiguous. Use the precise terms.
Business logic as a requirement	SR	"Business logic" is a code term, not a requirements term.
Functionality	`SR` / `TR`	Too broad; not unambiguously verifiable.
Feature (loose use)	Feature (SAFe context) or SR (the canonical RENAR term)	Ambiguous without a frame of reference.
Wish / "nice-to-have"	(never)	A contractual document is not written this way.
Epic as a requirement	BR (business level) or Portfolio Epic (SAFe)	An epic is a unit of planning, not a requirement.

"Test it by hand"	spot-check (Core Rule 5) or a manual TC with `tc-type: business`	Vague; no verifiable evidence.
"To finish later" (as a status)	`draft / review`	Not from the closed lifecycle list.
TODO in `frontmatter`	a backward entry in the `ADAPT` (if about a requirement) or a task in the tracker (if about implementation)	Open questions live in the right artifact, not in a field comment.

On such findings in project-local artifacts, raise a substrate-side warning (`pre-commit` in `git` , a validation rule in the document store).

5. Glossary versioning

The glossary is a standalone document with its own version. A change to a canonical term is a major-version bump (1.0 → 2.0) and a migration scenario for all project artifacts.

Current version: 1.0-reconciled (phase-1.5 reconciliation with `standard/04-terms.md §4.14.1`).

5.1 Open questions — closed (phase 1.5, 2026-05-16)

Four open questions from the earlier draft were closed by reconciliation with `standard/04-terms.md §4.14.1` :

#	Was an open question	Outcome	Source
1	Canonical language: latin (`BR` / `SR`) or Russian (БТ/ СТ)?	**Canon is latin**; Russian only in the UI projection (§3.5)	`standard/04 §4.13.3` + `reference/06-ru-style-guide.md §1.3`
2	`TM` as a separate label or an `SR` refinement?	**`SR` with `level: module`** — a refinement, not a separate artifact type	`standard/04 §4.14.1` (`TM` deprecated) + §2.1.1
3	AIC ← AAC / AIA / AIC?	**`SPEC-AI`** (v1.0 canon); AIC is a deprecated label	`standard/04 §4.14.1` + §2.1.1
4	`INT-TC` a separate type or a naming convention?	**`TC` with `tc-type: contract`** — a refinement, not a separate type	`standard/04 §4.14.1` (`INT-TC` deprecated) + §2.1.1

5.2 Reconciliation history

Date	Version	Change
2026-05-16	1.0-reconciled	Phase-1.5 reconciliation with `standard/04-terms.md §4.14.1` . Deprecated labels moved from the `§2.1` table to `§2.1.1` ; `QG` names aligned with the v1.0 canon in §2.4; deprecated names → `§2.4.1` . Mapping tables §3.1 and §3.2 updated. Four open questions closed (§5.1). Task: `ru-reconcile-glossary-vs-standard` .

(early drafts)	1.0	Draft fill-in during phase 7; had open questions and divergences from `standard/04 §4.14.1`. Commit history recorded; replaced by the 1.0-reconciled release.

5.3 Cross-references

- **EN Style Guide** (`reference/en/06-en-style-guide.md`) — EN editorial rules for normative text; §1.9 fixes the canonical term list alongside this glossary. On a conflict, editorial-pass wording priority belongs to the Style Guide.
- **Canonical definitions** (`standard/04-terms.md`); §4.14.1 — the deprecated → canon mapping.

RENAR Glossary 1.0-reconciled (EN) — part of `reference/`. See also 02-schemas.md, 03-ai-risk-register.md, 06-en-style-guide.md.

Schemas (formal)

> ***Purpose:*** *machine-readable YAML frontmatter schemas for all RENAR artifact types. Used by substrate-native validators to check conformance.* ***Normative structure definitions*** *live in* `standard/06`, `standard/07`, `standard/08`, `standard/09`. *Example validation:* `node scripts/validate-schema-examples.js`. *This document is a reference (informative lookup).*

1. Common frontmatter (all requirement and SPEC artifacts)

Fields common to BR/SR/TR/SPEC-* (canonical v1.0). Legacy types `UIC` / `AIC` / `INT-SR` / `TS` are deprecated in v1.0 (standard/04 §4.14.1).

```
# Identity
id: "<TYPE>-NN[.N]"
title: "<short, descriptive>"
type: BR | SR | TR | SPEC-ARCH | SPEC-API | SPEC-DATA | SPEC-INT | SPEC-PROC |
SPEC-UI | SPEC-AI | SPEC-SEC | SPEC-OPS | SPEC-TEST | SPEC-DOC
slug: "<kebab-case>"

# Scope
level: system | subsystem | module
scope: { system: "<system-id>", subsystem: "<subsystem-id>" }   # subsystem=null
if level=system

# Lifecycle (verified — BR/SR; frozen — ADAPT §7; ready/passing/failing — TC §8;
see standard/04 §4.6)
status: draft | review | approved | verified | deprecated | obsolete | frozen
priority: must | should | could       # MoSCoW; SAFe — via WSJF (BR-specific)

# Provenance (conditional, standard/07 §7.4.1): ADAPT is reactive.
# Findings present → source.adapt + adapt-section mandatory. No findings →
adversarial-review-ref mandatory.
# source.tz-section — always mandatory.
source:
  adapt: "ADAPT-NNN"                  # conditional
  adapt-section: "Forward §N.N"       # mandatory if adapt present
  tz-section: "§N.N"                  # always mandatory
  adversarial-review-ref: "AR-NNN"    # mandatory when adapt omitted → AR
(standard/07 §7.4.6)
  document-ref: "<link>"              # pinned revision of source document
  implementation-originated: {}       # conditional; see §1.1 (standard/06 §6.13)

# Hierarchy
parent: { id: "<parent-id>", ref: "<link>" }   # id required for SR (→BR), TR
(→SR); optional for BR
```

```
children: []                          # auto-derived

# Link to SPEC (graph)
constrained-by: []                    # SR → SPEC-* (typed edges)
implements-spec: []                   # TR → SPEC-*
depends-on: []                        # between SPEC

# Verification
verified-by: []                       # auto-derived; TC IDs
verifies-business-goal: ""            # optional

# AI provenance (RENAR-4+ mandatory for approved)
ai-provenance:
  generated-by: "<vendor>-<model>@<date>"
  prompt-template: "<template-path>@<version>"
  context-tokens: integer
  output-tokens: integer
  generation-time-ms: integer
  generated-at: "<ISO-8601>"
  human-edits: boolean                # true required for approved

# AI cost budget (optional)
ai-budget: { context-tokens-target: integer, context-tokens-actual: integer,
output-tokens-target: integer, output-tokens-actual: integer, generation-time-
target-ms: integer }

# Replacement + schema versioning
replaces: "<old-id>"
replaced-by: "<new-id>"
deprecated-date: "<ISO date>"
schema-version: "1.0"
```

1.1 `source: implementation-originated` — the implementation-originated requirement

A narrow legal origin class for BR / SR / SPEC (standard/06 §6.13): an **internal technical detail** added during implementation (a defensive check, internal validation, logging, an edge-case branch). Client-observable behaviour MUST NOT be covered by this class — it passes only through the contractual contour (ADAPT / ACTZ with signatures).

```
source:
  tz-section: "§N.N"                  # always mandatory (here too)
  implementation-originated:
    change-unit-ref: "<link to the implementation change unit>"   # mandatory —
provenance
    rationale: "<why the functionality was deemed necessary>"     # mandatory
    human-approval:                                               # mandatory; V6
      approved-by: "<human supervisor name>"
      role: "<role>"
      approved-at: "<ISO-datetime>"
    observable-by-client: false       # mandatory; true → non-conformance
```

```
(standard/06 §6.13.4)
    covering-tc: "TC-NN"              # mandatory; the TC exists and passes
before the change is merged
    mutation-check:                                                  # mandatory —
compensates for the absent red history
      mutants-killed: integer         # >= 1; otherwise the TC is not evidence
(standard/09 §9.18.2)
      report-ref: "<link to the mutation-check report>"
```

Rule	Level
`human-approval` is mandatory; an agent MUST NOT legalise its own addition	normative
`covering-tc` exists and passes **before** the change is merged	hook-enforced
`covering-tc` MUST NOT have a red history; a killed mutant is required instead (`mutation-check.mutants-killed >= 1`)	hook-enforced
`observable-by-client: true` together with `implementation-originated` is a non-conformance (an obligation to the client MUST NOT arise from code)	normative
The share of artifacts of this class is a counter in the drift metrics (standard/12 §12.3)	normative

2. BR — Business Requirement

```
# Extends common §1, additionally:

level: system | subsystem             # BR at module level is prohibited
(standard/06 §6.4)
scope: { system: "<system-id>", subsystem: "<subsystem-id>" }

# Cross-level link: subsystem BR → system BR (standard/06 §6.8.2)
implements:                             # array; substrate-agnostic reference
  - { id: BR-NN, scope: { system: "<system-id>" }, rationale: "<short>" }   #
rationale optional
implemented-by: []                      # auto-derived (reverse edge; not written
by the author)

business-context:
  stakeholder: "<role>"
  business-goal: "<short statement>"
  kpi-impact:
    - { kpi: "<name>", direction: increase|decrease, target: "<measurable>" }

# business-outcome — required for QG-4
business-outcome:
  measurement-type: kpi | survey | observation | usage
  kpi-name: "<KPI>"
  measurement-method: "<how>"
```

```
  baseline-value: number
  baseline-measured-at: "<ISO date>"
  target-value: number
  target-met-by: "<ISO date>"
  current-value: { value: number, measured-at: "<ISO date>", achievement: "
<percent>" }

prioritization: { framework: WSJF|RICE|MoSCoW, wsjf-score: number, prioritized-
at: "<ISO date>", prioritized-by: "<role>" }

data-classification:
  contains-pii: boolean
  contains-financial: boolean
  contains-health: boolean
  contains-children-data: boolean
  retention-days: integer
  data-residency: ["RU" | "EU" | "US" | ...]

compliance:
  - { standard: "ISO 27001:2022", control: "<id>", rationale: "..." }
  - { standard: "GDPR", article: "Art.NN" }
  - { standard: "ФЗ-152", article: "ст.NN" }

ai-act: { risk-class: prohibited|high|limited|minimal, rationale: "<reason>",
high-risk-domain: boolean }
```

Fields `implements[]` / `implemented-by[]` — normative rules

Rule	Level
`implements[]` required when `level: subsystem` AND the parent system has ≥1 approved BR	recommended v1.0; mandatory v1.1+
The target BR MUST be in status `approved` or higher at the moment this BR is approved	hook-enforced
Cycle detection: the `implements` chain MUST NOT form cycles	hook-enforced
`implements[]` is **not** a parent edge; the ban on multiple parents (standard/06 §6.8.3) applies to SR/TR, not to BR	normative
Deprecate target BR → cascade-warning for all `implemented-by` (not cascade-deprecate)	hook-enforced
Cross-substrate syntax: `id + scope.system` is substrate-independent	normative
Cardinality: array (0..N)	normative

The enforcement gate is `scripts/check-implements-edge.js`. The `implemented-by` field is auto-derived; manual entry is prohibited.

3. SR — System Requirement

```
# Extends common §1, additionally:

parent:
  id: "BR-NN"                          # required

# ADAPT source — via the canonical source.adapt / source.adapt-section (§1).
# There is no separate derived-from-adapt field (standard/06 §6.6.2).

constrained-by:                        # typed edges to SPEC-*
  - "SPEC-UI-NN"
  - "SPEC-API-NN"
  - "SPEC-DATA-NN"
  - "SPEC-SEC-NN"

quality-characteristic:                # ISO/IEC 25010:2023 (9 characteristics;
interaction-capability ← usability, flexibility ← portability in 25010:2011;
safety — new in 25010:2023)
  - functional-suitability | performance-efficiency | compatibility |
interaction-capability | reliability | security | maintainability | flexibility |
safety

# Inherited from parent BR (where applicable): data-classification, compliance,
ai-act
```

4. TR — Task Requirement

```
# Extends common §1, additionally:

parent:
  id: "SR-NN"                          # required

implements-spec: []                    # SPEC-* implemented by this task
estimated-effort: "<short statement>" # optional, free-form
```

5. SPEC-* common schema

All 11 SPEC types share a common structure (§1) plus the following SPEC-specific fields:

```
type: SPEC-ARCH | SPEC-API | SPEC-DATA | SPEC-INT | SPEC-PROC | SPEC-UI | SPEC-AI
| SPEC-SEC | SPEC-OPS | SPEC-TEST | SPEC-DOC

referenced-by: []                      # auto-derived
```

```
depends-on: []                          # SPEC this one depends on
compliance-refs: []                     # ISO / GDPR / Ф3-152 / AI Act / NIST AI
RMF
```

Mandatory body sections: `## Purpose`, `## Scope`, `## <Type-specific sections — see §6>`, `## Link to requirements`, `## Link to other SPEC`, `## Verification`, `## Open questions`.

6. SPEC type-specific extensions

Type-specific fields for the 11 SPEC types. Industry references are in `standard/14`. Legacy replacements: `UIC` → SPEC-UI, `AIC` → SPEC-AI, `INT-SR` → SPEC-INT.

6.1 SPEC-ARCH, SPEC-API, SPEC-DATA, SPEC-INT, SPEC-PROC

```
# SPEC-ARCH
arch-style: monolith | microservices | modular-monolith | serverless | hybrid
deployment-model: cloud | on-prem | hybrid | edge
tech-stack: { languages: [], frameworks: [], data-stores: [], message-brokers: []
}
quality-attributes: [{ name: latency, target: "p95 < 200ms" }, { name:
availability, target: "99.9%" }]

# SPEC-API
api-style: rest | graphql | grpc | websocket | async-events
api-version: "v1.2.0"
versioning-strategy: url-path | header | query-param | content-negotiation
authentication: bearer-jwt | api-key | oauth2 | mtls | none
rate-limits: [{ endpoint: "*", limit: "1000/min/key" }]
contract-file: { format: openapi-3.1 | asyncapi-2.6 | proto3, location:
"contracts/<name>.yaml" }

# SPEC-DATA
data-style: relational | document | graph | columnar | hybrid
storage-engine: postgresql | mysql | couchdb | mongodb | clickhouse | ...
schema-version: "1.4.0"
pii-classification: [{ entity: User, fields: [email, phone], level: PII-high }]
retention-policies: [{ entity: Order, period: "7 years", basis: "tax law" }]
migration-strategy: forward-only | reversible | dual-write

# SPEC-INT
integration-pattern: request-response | event-driven | message-queue | webhook |
file-transfer
direction: outbound | inbound | bidirectional
counterparty: { system: "<external-name>", contract-owner: "<team-or-vendor>",
contract-ref: "<external-spec-url>" }
sla: { availability: "99.5%", latency-p95: "500ms", fallback: "queue + retry;
manual reconciliation after 24h" }
idempotency: guaranteed | best-effort | none
```

```
# SPEC-PROC
process-style: bpmn | state-machine | saga | choreography | orchestration
state-count: integer
participants: [{ role: customer, system: client-portal }, { role: agent, system:
back-office }]
sla: { end-to-end: "2 business hours" }
compensation: defined | not-applicable | manual
```

6.2 SPEC-UI, SPEC-AI, SPEC-SEC, SPEC-OPS

```
# SPEC-UI
ui-platform: web | mobile-ios | mobile-android | desktop | tv | embedded
target-users: [{ role: end-customer, persona: "ADAPT-NNN §X.Y" }]
design-system: "<reference-or-internal>"
accessibility-level: WCAG-A | WCAG-AA | WCAG-AAA
i18n: required | not-required
mockup-links: [{ tool: figma, url: "<link>", version: "v3" }]
baseline-images: ["ai-concepts/baselines/SPEC-UI-NN-screen-01.png"]

# SPEC-AI (judge-model.vendor ≠ production-model.vendor — normative)
ai-pattern: rag | fine-tuning | prompt-engineering | tool-use | multi-agent |
embedding-only
production-model: { vendor: anthropic|openai|google|local, model: "<name>",
version: "<exact>" }
judge-model: { vendor: "<different-vendor>", model: "<different-model>" }
context-strategy: { embedding-model: "<model>", chunk-size: integer, chunk-
overlap: integer, vector-store: pinecone|weaviate|pgvector|qdrant }
eval-strategy: { metric: accuracy|f1|rouge|custom-rubric, threshold: number,
baseline-dataset: "<path>" }
cost-budget: { tokens-per-request-target: integer, tokens-per-request-ceiling:
integer, monthly-budget-usd: number }

# SPEC-SEC
security-domains: [authentication, authorization, data-protection, audit,
secrets-management]
auth-model: { authn: jwt-bearer|oauth2-pkce|mtls|passkey, authz: rbac|abac|relbac
}
data-classification: [{ class: PII-high, fields: [...] }, { class: PCI, fields:
[...] }]
threat-model-method: STRIDE | PASTA | OCTAVE
compliance: [ISO-27001, GDPR, Ф3-152, PCI-DSS-4]

# SPEC-OPS
deployment-style: kubernetes | vm | serverless | docker-compose | bare-metal
environments:
  - { name: dev, purpose: development, scale: minimal }
  - { name: staging, purpose: integration-testing, scale: half-prod }
  - { name: prod, purpose: production, scale: full }
slo: { availability: "99.9%", error-budget-month: "43m", latency-p95: "300ms" }
observability: { logs: elastic|loki|cloudwatch, metrics:
```

```
prometheus|datadog|cloudwatch, traces: jaeger|tempo|x-ray }
disaster-recovery: { rto: "<duration>", rpo: "<duration>" }
```

6.3 SPEC-TEST — benches and data

A test bench is not a deployment environment but a construction of proof (`standard/08 §8.3.2` , §8.5.10). Datasets are described **on both sides** of every integration; the expected result often lives outside our system, so reconciliation rules are a mandatory part of the schema.

```
# SPEC-TEST
benches:                              # bench topology: nodes, versions, what is
live / what is emulated
  - name: "<bench-id>"
    nodes: [{ name: "<node>", version: "<version>", mode: live | emulated }]
    notes: "<bench limitations>"

counterparties:                       # one entry per integration (exactly one
per SPEC-INT)
  - integration: SPEC-INT-NN
    representation: real | sandbox | emulator
    endpoint: "<substrate-native pointer>"
    fidelity-rationale: "<why the representation answers like the real system>"

datasets:                             # on both sides of every integration
  - name: "<dataset-id>"
    side: ours | counterparty
    integration: SPEC-INT-NN          # null for a dataset outside an integration
    volume: "<volume>"
    origin: synthetic | anonymized-production | client-provided
    anonymized: boolean
    contains-real-client-data: boolean   # true ⇒ client-signature is REQUIRED
    retention: "<retention period>"

reconciliation-rules:                 # reconciling results against data held by
foreign systems
  - name: "<rule-id>"
    integration: SPEC-INT-NN
    expected-result-source: "<where the expected result lives when it is not in
our system>"
    match-keys: []
    tolerance: "<reconciliation tolerance>"

run-config:
  mocked: [SPEC-INT-NN]               # what is mocked
  live: [SPEC-INT-NN]                 # what runs live
  run-order: []                       # order of the run
  seed-mechanism: "<how the bench state is prepared>"

client-signature:                     # REQUIRED when real / personal client data
is used
  signed-by: "<name>"
```

```
  role: "<role>"
  organization: "<client-org>"
  signed-at: "<ISO-datetime>"
  signature-ref: "<link>"
```

Rule	Level
At least one `datasets[]` entry with `contains-real-client-data: true` (or `anonymized: false`) ⇒ `client-signature` MUST be filled in; otherwise — **fatal**	hook-enforced
`counterparties[]` MUST hold exactly one entry per integration represented on the bench	hook-enforced
`reconciliation-rules[].expected-result-source` MUST be non-empty when the expected result lives outside our system	normative
A version increment of a SPEC-TEST invalidates `verified` on the TCs that reference it through `environment-ref` (standard/10 §10.5.4)	hook-enforced

6.4 SPEC-DOC — delivered documentation

SPEC-DOC describes the composition of the delivered result: the documents the client receives as part of the delivery (`standard/08 §8.5.11`). The boundary with SPEC-OPS is membership in the delivery: an administrator manual is always SPEC-DOC, a runbook is always SPEC-OPS.

```
# SPEC-DOC
deliverables:                          # one entry per document
  - name: "<document-id>"
    kind: user-manual | admin-manual | training-materials | other
    audience: "<reader role>"
    format: pdf | html | markdown | docx | other
    language: "<delivery language>"

required-sections:                     # mandatory sections — a requirement, not
the vendor's discretion
  - deliverable: "<document-id>"
    sections: ["<mandatory section heading>"]

traces-to: []                          # BR / SR / SPEC whose behaviour the
document describes

version-binding:
  rule: matches-system-version | matches-release-tag | manual
  system-version-ref: "<substrate-native link to the system version>"

acceptance-criteria:                   # what the document is accepted against;
checked by the doc-lint
  - criterion: "<checkable assertion>"
    checked-by: TC-NN                  # a TC with automation.kind: static
(standard/09 §9.8)
```

Rule	Level
The doc-lint is mandatory: a SPEC-DOC MUST have at least one TC with `automation.kind: static`; without it the SPEC-DOC is unverifiable and **does not pass QG-2** (standard/09 §9.8)	hook-enforced
`required-sections[]` MUST be non-empty for every `deliverables[]` entry	hook-enforced
`traces-to[]` MUST be non-empty: the document describes behaviour stated in the requirements; the doc-lint checks coverage of roles and scenarios	normative
Substantive conformance of the text to implemented behaviour is OPTIONAL, via a judge agent (P7 / `eval`); no separate mechanism is introduced	normative

7. ADAPT schema

ADAPT is a separate artifact (`standard/07`). It is reactive: it exists only when there are findings from the adversarial review of the TZ (§7.4.1). On a "no findings" verdict, no ADAPT is created; the review outcome is issued as an AR in either case (§7.1 below), and artifacts reference it via `<artifact>.source.adversarial-review-ref`.

```
# Identity
id: ADAPT-NNN
title: "Adaptation of TZ <name>"
type: ADAPT
trigger-stage: import-tz | decompose-br | decompose-sr | spec | tc   # trigger
stage (standard/07 §7.4.1.4)

# Source
source-tz: { id: TZ-YYYY-NNN, signed-date: "<ISO-date>", signed-by-client: "
<name-role>", document-version-ref: "<substrate-native version identifier>" }   #
V5 pin
parent-adapt: { id: ADAPT-NNN, delta-tz: TZ-YYYY-NNN }   # for delta-ADAPT

# Supersession (standard/07 §7.6.4) — only for superseding-ADAPT
supersedes: ADAPT-MMM                                   # reference to the
superseded ADAPT
superseded-by: ADAPT-NNN                                # auto-derived; on the
superseded one
supersession-rationale: "<contradicting BR/SR/SPEC + source>"   # mandatory if
supersedes present

# Lifecycle (subset of §1: ADAPT does not use verified/deprecated/obsolete;
superseded — terminal on supersession, §7.6.4)
# The client-ready state is REMOVED: putting questions to the client is a matter
for the ACTZ (§7.13), not an ADAPT state (standard/10 §10.8.1).
status: draft | review | asked | answered | approved | frozen | superseded
created: "<ISO-date>"
last-updated: "<ISO-date>"

# Approval (required for approved)
```

```
approval:
  # No client-signature: an ADAPT is an internal interpretation (standard/07
§7.5).
  # The client's obligations are carried by an ACTZ (§7.13) with the dual
signature.
  architect-signature: { signed-by: "<name>", role: architect, signed-at: "<ISO-
datetime>" }

# Auto-derived
generates-requirements: []
generates-specs: []
open-questions-count: integer         # MUST be 0 for approved
resolved-questions-count: integer

# AI provenance
ai-provenance: { generated-by: "<vendor>-<model>@<date>", prompt-template: "
<template-path>@<version>", context-tokens: integer, output-tokens: integer,
human-edits: boolean }
```

Backward entries inside the body:

```
id: B-NNN
category: contradiction | gap | hidden-assumption | feasibility | regulatory |
terminology | scope
status: open | asked-to-client | answered | resolved | revised | frozen
tz-section: "§N.N"
description: "..."
asked-to-client: "<ISO-date>"
client-answer:
  signed-by: "<name>"
  signed-at: "<ISO-datetime>"
  channel: email | docusign | zoom-transcript | written-letter
  text: "..."
resolution: "..."                     # how the answer was integrated into
Forward
decided-in: "ACTZ-NNN §M"             # mandatory for status=resolved; a clause
of a signed ACTZ (§7.2)
```

A finding moves to `resolved` only once the decision on it has been **put to the client and signed**: `decided-in` points at clause `§M` of an ACTZ in status `signed` (standard/07 §7.13.2). Referencing an ACTZ in status `draft` is forbidden (standard/07 §7.13.4).

7.1 AR schema — the adversarial-review record

AR (`standard/07 §7.4.6`) is an evidence record capturing the fact and the outcome of the mandatory adversarial review. It is issued in **both** outcomes; on `no-findings` it is itself the evidence referenced by `<artifact>.source.adversarial-review-ref`. It is not a requirements artifact and belongs to no closed list.

```
# Identity
id: AR-NNN                              # sequential within the parent TZ;
immutable
type: AR
tz-ref: TZ-YYYY-NNN                     # the (delta-)TZ under review
trigger-stage: import-tz | decompose-br | decompose-sr | spec | tc   # aligned
with ADAPT.trigger-stage

# Reviewer (isolation: model != primary agent's model, standard/07 §7.10.2)
reviewer: { vendor: "<provider>", model: "<model-id>" }

# Verdict
verdict: findings-present | no-findings
produces-adapt: [ADAPT-NNN]             # mandatory non-empty when findings-
present; empty when no-findings

# Lifecycle
status: draft | issued | superseded
superseded-by: AR-NNN                   # mandatory when status=superseded

# Signature (V6; mandatory for issued)
signature: { author: "<reviewer-id>", timestamp: "<ISO-8601>" }
```

7.2 ACTZ schema — the TZ clarification protocol

ACTZ (`standard/07 §7.13`) is an artifact of the **contractual contour**: a batch of questions, proposals and decisions put to the client and signed by both parties. The boundary with ADAPT is drawn by audience: what is shown to the client and approved is an obligation; what is not shown is interpretation. Cardinality: TZ : ACTZ = 1 : 0..N; ADAPT : ACTZ = 1 : 1..N.

```
# Identity
id: ACTZ-NNN                            # sequential within the parent TZ;
immutable
title: "TZ Clarification Protocol No. N"
type: ACTZ
tz-ref: TZ-YYYY-NNN                     # mandatory; the TZ the protocol belongs to

# Findings closed (conditional)
resolves:                               # B-NNN entries in an ADAPT that the
protocol closes
  - { id: B-NNN, adapt: ADAPT-NNN }     # absent on a client-initiated ACTZ
(standard/07 §7.13.2)

# Decisions (mandatory, non-empty)
decisions:
  - number: "§M"                        # stable clause number; the target of
decided-in
    statement: "<decision in the language of obligations: 'the button is named
X'>"
    tz-section: "§N.N"                  # the TZ section being clarified
```

```
# TZ annexes (conditional; standard/07 §7.13.5)
annexes:
  - { name: "<annex>", version: "<new version>", document-ref: "<link>" }

# Lifecycle
status: draft | sent | signed | superseded
superseded-by: ACTZ-NNN              # mandatory if status=superseded

# Signatures (mandatory for signed; V6)
client-signature: { signed-by: "<name>", role: "<role>", organization: "<client-
org>", signed-at: "<ISO-datetime>", signature-ref: "<link>" }
vendor-signature: { signed-by: "<name>", role: "<role>", signed-at: "<ISO-
datetime>" }
```

Rule	Level
`decisions[].number` is stable and immutable: `B-NNN.decided-in` and `AT.verifies[]` point at it	normative
A signed ACTZ is immutable; a correction is made only by a new ACTZ carrying `superseded-by` on the previous one	hook-enforced
Referencing an ACTZ in status `draft` from `decided-in` is forbidden	hook-enforced
`resolves[]` is absent on a client-initiated protocol; once signed, the decision MUST be reflected in the ADAPT	normative
A TZ annex is not addressable on its own: its new version is approved through `annexes[]` under the same two-sided signature	normative
Effective TZ = the initial TZ (with annexes) + all signed ACTZ; the later signed document prevails (standard/07 §7.14)	normative

8. TC — Test Case

```
# Identity
id: "TC-NN[.N]"
title: "<descriptive>"
type: TC
slug: "<kebab-case>"

# Classification
tc-type: business | ux | system | contract | eval | security   # business — the
former acceptance (standard/09 §9.5)
negative: boolean                    # true for the paired negative TC

# Scope
level: system | subsystem | module
scope: { system: "<system-id>", subsystem: "<subsystem-id>", module: "<module-
```

```
id>" }

# Lifecycle
status: draft | ready | passing | failing | obsolete

# Verification mapping (≥1)
verifies:
  - { id: "<requirement-id>", ref: "<link>", requirement-version: "<version-ref>"
}   # V5 pinning (standard/03 §3.3.5)

# Pair link (mandatory if negative=false and a pair exists)
paired-with: ["<TC-id>"]

# Task binding (optional; standard/09 §9.19.7)
verifies-tr: "TR-NN"                  # the task to whose scope the test is
narrowed
verifies-claims: []                   # the subset of the parent SR's claims
covered within that TR

# Bench and data (mandatory; standard/09 §9.3, standard/08 §8.5.10)
environment-ref: "SPEC-TEST-NN"       # conditional: mandatory if
automation.kind: dynamic.
                                      # Does not apply to static checks (the doc-
lint of §9.8, the
                                      # structural TC of §9.8.1): the analyzer
works over artifacts

# Automation
automation:
  status: automated | manual-pending
  kind: dynamic | static                    # mandatory; static — the runner is a
static analyser
  location: "<path-to-implementation>"      # mandatory if automated
  runner: pytest | jest | go-test | playwright | vlm-judge | ragas | pact | other
  manual-pending-until: "<ISO date>"        # mandatory if manual-pending
  manual-pending-reason: "<why>"

# Red history (standard/09 §9.18.2) — the condition for counting a TC as evidence
red-history:
  fixing-run:                               # the fixing run, performed before
implementation
    date: "<ISO timestamp>"
    result: fail                            # MUST be red; pass → a signal to
investigate, not a success
    run-ref: "<link>"
  green-transition:                         # the recorded red → green transition
    date: "<ISO timestamp>"
    run-ref: "<link>"
  inherited-from: "TC-NN"                   # conditional: inheritance for tasks
that do not change behaviour
  not-applicable-reason: implementation-originated   # conditional: this class
only; then a killed mutant is mandatory
  mutation-check: { mutants-killed: integer, report-ref: "<link>" }   # mandatory
when not-applicable-reason is set
```

```
# Execution (mandatory if tc-type=ux | eval; judge.vendor ≠ production model
vendor — see §6.2 SPEC-AI, §9)
judge: { vendor: "<provider>", model: "<model-id>", prompt-template: "<template-
path>@<version>" }
baseline: { artifact: "<pointer>", perceptual-diff-threshold: float, metric-
thresholds: {} }

# Last run (auto-managed; bot-only)
last-run:
  date: "<ISO timestamp>"
  result: pass | fail | skipped | n/a
  runner-id: "<runner@version>"
  run-ref: "<link>"
  requirement-version: "<version-ref>"
  judge-report: "<for ux/eval>"

# Replacement / obsolescence
obsolete-pending: boolean             # true on detected delta-TZ invalidation
replaces: "<old-id>"
replaced-by: "<new-id>"
obsoleted-date: "<ISO date>"

# Inherited
ai-provenance: { ... }                # see §1
```

`tc-type: business` is the canonical name; the former `acceptance` is not used in v1.0. Acceptance against the effective TZ is performed not through TC but through AT (§8.1).

8.1 AT schema — the acceptance test of the contractual contour

AT (`standard/09 §9.19`) is derived **exclusively** from the effective TZ (standard/07 §7.14): the initial TZ with its annexes plus every signed ACTZ (§7.2). AT checks conformance to the **contract**, not to the interpretation, and is created by an isolated agent that MUST NOT be given access to the internal contour (ADAPT, BR / SR / SPEC, TC, code).

```
# Identity
id: AT-NN                             # immutable
title: "<short, descriptive>"
type: AT
negative: boolean                     # mandatory; pos/neg pairing — as for TC
(standard/09 §9.7)

# Verification target (mandatory; closed list of references)
verifies:
  - "TZ §N"                           # a section of the effective TZ
  - "ACTZ-NNN §M"                     # a clause of a signed protocol
# A reference to an internal artifact (BR / SR / SPEC / TC) is fatal (standard/09
§9.19.2)

tz-version: "<effective TZ revision>" # mandatory; the revision the AT was
```

```
derived from

# Provenance of the isolated agent (mandatory)
generator:
  vendor: "<provider>"                # MUST differ from the primary agent's
model (mirroring P7)
  model: "<model-id>"
  internal-contour-access: false      # mandatory; confirmation of no access to
the internal contour
  generated-at: "<ISO-8601>"

# Lifecycle
status: draft | ready | passing | failing | obsolete

# The acceptance-trial environment and dataset (mandatory; standard/09 §9.19.3,
§8.5.10).
# NOT filled in by the generator: the isolated agent does not see internal
artifacts —
# the Architect or the runner sets this AFTER generation, leaving isolation
intact.
environment-ref: SPEC-TEST-NN

# Automation (mandatory; execution only by an automated runner)
automation:
  status: automated | manual-pending
  kind: dynamic | static
  location: "<path-to-implementation>"
  runner: "<runner>"

# Last run (runner-managed; bot-only)
last-run:
  date: "<ISO timestamp>"
  result: pass | fail | skipped | n/a
  runner-id: "<runner@version>"
  run-ref: "<link>"
  tz-version: "<effective TZ revision at the time of the run>"
```

The body of an AT MUST contain a section with the **verbatim quotation** of the effective-TZ clause under test (`tz_text`) next to the verification steps (standard/09 §9.19.3).

Rule	Level
`verifies[]` contains only `TZ §N` and `ACTZ-NNN §M`; a reference to an internal artifact is **fatal**	hook-enforced
`generator.internal-contour-access: true` is **fatal**: isolation is the substance of the mechanism, not hygiene	hook-enforced
`generator.model` differs from the primary agent's model	hook-enforced
ATs are regenerated before every trial; `tz-version` MUST match the current revision of the effective TZ, otherwise the trials are blocked	hook-enforced

The product is not submitted for hand-over until every AT is in status `passing` (standard/10 §10.4.3)	normative

9. Validation rules (cross-field)

Rules not expressible in pure JSON Schema; they require a custom validator. The "Formal check" column gives an executable predicate or a reference to a ready-made KG query (reference/05 §5/§6).

Rule	Description	Formal check
ID immutable	When a file changes, the `id` field does not change.	`diff(prev.id, curr.id) == ∅`
`parent` exists	For SR — the parent BR exists and is in status ≥ `approved`.	`status(BR[SR.parent.id]) ≥ approved`; orphans — 05 §6.1
`source.adapt` approved	For BR/SR/SPEC — the ADAPT in `source.adapt` is in status `approved` / `frozen`.	`status(ADAPT[art.source.adapt]) ∈ {approved, frozen}`
`verified-by` consistency	TCs in `verified-by` have `verifies[].id` = this artifact.	`∀ tc ∈ art.verified-by: art.id ∈ tc.verifies[].id`
`requirement-version` lock	`TC.last-run.requirement-version` = `verifies[].requirement-version`.	`tc.last-run.requirement-version == tc.verifies[].requirement-version`; stale — 05 §4.4
`source.adapt` for BR/SR/SPEC (conditional)	Canonical ADAPT source when findings are present; on a "no findings" verdict — `source.adversarial-review-ref` (standard/07 §7.4.1). TR — via parent SR (standard/06 §6.6.2).	`art.type ∈ {BR, SR, SPEC-*} ⇒ art.source.adapt ≠ null ∨ art.source.adversarial-review-ref ≠ null`
SPEC-AI requires ai-act	For an AI artifact, `ai-act.risk-class` is mandatory.	`art.type == SPEC-AI ⇒ art.ai-act.risk-class ≠ null`
Data residency consistency	RU in `SR.data-classification.data-residency` ⇒ the same in the parent BR.	`'RU' ∈ SR.…data-residency ⇒ 'RU' ∈ BR[SR.parent].…data-residency`
Compliance hierarchy	`SR.compliance ⊆ parent BR.compliance` (or explicit justification).	`SR.compliance ⊆ BR[parent].compliance ∨ exists(extension-justification)`
TC `automated` requires location	`automation.status: automated` ⇒ `automation.location` non-empty.	`tc.automation.status == 'automated' ⇒ tc.automation.location ≠ ''`
Negative TC mandatory	For every normative assertion — a TC with `negative: true`.	`∀ assertion ∈ art: ∃ tc(negative: true)` (standard/09 §9.7)

ADAPT open-questions == 0 for approved	Approval is blocked while there are `open` / `asked-to-client` / `answered` / `revised` backward entries: every finding MUST be `resolved` (standard/07 §7.4.5, standard/10 §10.8.2).	`adapt.status == approved ⇒ count(backward[status ∈ {open, asked-to-client, answered, revised}]) == 0` (05 §4.7)
Supersession correct	`supersedes` ⇒ non-empty `supersession-rationale` and a symmetric `superseded-by` on the target; no dangling `source.adapt` on a `superseded` ADAPT (standard/07 §7.6.4, standard/10 §10.8.5).	`adapt.supersedes ≠ null ⇒ adapt.supersession-rationale ≠ '' ∧ ADAPT[adapt.supersedes].superseded-by == adapt.id ; ∄ art: art.source.adapt = X ∧ status(ADAPT[X]) == superseded`
Judge isolation (SPEC-AI)	`judge.vendor ≠ production-model.vendor` .	`tc.judge.vendor ≠ SPEC-AI[tc.verifies].production-model.vendor`
SPEC depends-on acyclic	The `depends-on` graph between SPEC is a DAG.	cypher cycle-detection (05 §4.6): rows ≠ ∅ ⇒ violation
A `resolved` finding is decided by a signed ACTZ	A backward finding in status `resolved` MUST carry `decided-in: ACTZ-NNN §M` , and the target ACTZ MUST be in status `signed` (standard/07 §7.13.2).	`b.status == resolved ⇒ b.decided-in ≠ null ∧ status(ACTZ[b.decided-in]) == signed ∧ b.decided-in.§M ∈ ACTZ.decisions[].number`
A `signed` ACTZ is signed by both parties	Both signature fields are filled in; a signed protocol is immutable.	`actz.status == signed ⇒ actz.client-signature ≠ null ∧ actz.vendor-signature ≠ null`
AT references the contract only	`AT.verifies[]` contains only `TZ §N` / `ACTZ-NNN §M` ; a reference to an internal artifact is fatal (standard/09 §9.19.2).	`∀ v ∈ at.verifies: v ~ /^(TZ §
AT is fresh against the effective TZ	`AT.tz-version` = the current revision of the effective TZ; a divergence blocks the trials (standard/09 §9.19.4).	`at.tz-version == effective-tz.version`
Red history is the condition for counting a TC	A TC counts as evidence at QG-2 only with a recorded red → green transition; the exception is the `implementation-originated` class with a killed mutant (standard/09 §9.18.2).	`tc.red-history.green-transition ≠ null ∨ tc.red-history.inherited-from ≠ null ∨ (tc.red-history.not-applicable-reason == implementation-originated ∧ tc.red-history.mutation-check.mutants-killed ≥ 1)`
`implementation-originated` is not client-observable	The class legalises an internal detail; client-observable behaviour MUST NOT pass through it (standard/06 §6.13.2).	`art.source.implementation-originated ≠ null ⇒ observable-by-client == false ∧ human-approval ≠ null ∧ mutation-check.mutants-killed ≥ 1`

A dynamic TC is bound to a bench	A TC with `automation.kind: dynamic` MUST carry an `environment-ref` to an existing SPEC-TEST: "the test passed" only means something against a specific bench and specific data (standard/09 §9.3). A missing field on a dynamic TC is **fatal**. Static checks (the doc-lint, the structural TC) require no bench.	`tc.automation.kind == 'dynamic' ⇒ tc.environment-ref ≠ null ∧ type(SPEC[tc.environment-ref]) == 'SPEC-TEST'`
Real client data is signed off by the client	A SPEC-TEST in which at least one dataset carries real or personal client data MUST carry a `client-signature`; volume and anonymization are the client's decision, not the vendor's (standard/08 §8.5.10). A missing signature is **fatal**.	`∃ d ∈ spec.datasets: d.contains-real-client-data == true ∨ d.anonymized == false ⇒ spec.client-signature ≠ null`
A SPEC-DOC is covered by the doc-lint	A SPEC-DOC MUST have at least one doc-lint TC (`automation.kind: static`); without it the SPEC-DOC is unverifiable and **does not pass QG-2** (standard/09 §9.8).	`spec.type == 'SPEC-DOC' ⇒ ∃ tc ∈ spec.verified-by: tc.automation.kind == 'static'`

10. Substrate isomorphism

Mapping for git (YAML frontmatter) ↔ document-oriented store (JSON document):

Field (canonical)	git (YAML frontmatter)	Document store (JSON doc)
`id`	`id`	`_id = <project>:<doc-type>:<slug>` , field `slug`
`type: BR`	`type: BR`	`level: "business"`
`type: SPEC-API`	`type: SPEC-API`	`doc_type: "spec_api"`
`parent.id`	`parent.id`	`parent`
`children`	(auto-derived)	`children` (auto-derived)
`status`	`status`	`status`
`priority`	`priority`	`priority`
`source.adapt`	`source.adapt`	`created_from_adapt`
`constrained-by[]`	`constrained-by`	`constrained_by` (subdoc array)
`verified-by[]`	(auto-derived)	`linked_tests`
`ai-provenance.*`	nested object	nested subdocument

`compliance`	`compliance` array	`compliance` subdoc
`data-classification`	nested object	nested subdoc
`business-outcome`	nested object	nested subdoc
`replaces` / `replaced-by`	string ID	`replaces` / `replaced_by`

Substrate-native field names MAY differ, but the **semantics and invariants are preserved** through capabilities V1–V6 (01-glossary.md §2.7).

11. Schema versioning

Every artifact has a `schema-version` field (semver). When the file version and the current schema do not match, the validator proposes a migration.

Change	Bump
New optional field	minor (1.0 → 1.1)
New mandatory field	major (1.0 → 2.0) + migration script
Field removal	major + migration script
Enum change	minor if addition, major if removal
Field rename	major + migration script

Current schemas version: 1.0.

12. JSON Schema fragment example (BR)

Key patterns (the full BR schema — `reference/schemas/br.json`, planned):

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://renar.tech/schemas/br.json",
  "type": "object",
  "required": ["id", "title", "type", "status", "priority", "source", "ai-
provenance"],
  "properties": {
    "id":     { "type": "string", "pattern": "^BR-[0-9]{2}(\\.[0-9]+)?$" },
    "title":  { "type": "string", "minLength": 5, "maxLength": 100 },
    "type":   { "const": "BR" },
    "status": { "enum": ["draft", "review", "approved", "verified", "deprecated",
"obsolete"] },
    "priority": { "enum": ["must", "should", "could"] },
    "source": { "type": "object", "required": ["tz-section"], "properties": {
"adapt": { "pattern": "^ADAPT-[0-9]{3}(-delta-[0-9]+)?$" } } },
    "ai-provenance": { "required": ["generated-by", "generated-at"],
"properties": { "generated-by": { "pattern": "^[a-z]+-[a-z0-9-]+@[0-9]{4}-[0-9]
```

```
{2}-[0-9]{2}$" } } }
  }
}
```

Equivalent JSON schemas for SR/TR/SPEC-*/TC/ADAPT are in `reference/schemas/` (planned).

Schemas reference RENAR 1.0 — see also 01-glossary.md, `standard/06` - `09` for normative definitions.

AI Risk Register for RENAR Projects

> ***Purpose:*** *a register of AI-specific risks for projects that use RENAR (where AI generates requirements, specifications, and tests). Based on ISO/IEC 23894:2023 (AI Risk Management, Annex A risk sources + Clause 6 process per ISO 31000) and NIST AI RMF 1.0. Normative mitigation hooks — standard/09 §9.13, standard/07 §7.10.*

This does not replace the organization's general security risk register. AI risks are a separate class because of the specifics of generation and the unpredictability of models.

1. Register structure

Each risk has:

```
id: AIR-NN                              # immutable
name: "<short name>"
category: hallucination | injection | drift | bias | sgnl-failure | data-quality
| adversarial | privacy
# enum value — the part before the parenthesis; the qualifier in parentheses is
optional (e.g. "sgnl-failure (process)")
severity: critical | high | medium | low
likelihood: high | medium | low
iso-23894-ref: "§N.N"
nist-rmf-function: govern | map | measure | manage
mitigations:
  - { mechanism: "<description>", enforced-by: "<who/what>", automated: true |
false }
status: active | mitigated | accepted | monitoring | out-of-scope
owner: "@role"
last-reviewed: "YYYY-MM-DD"
related: ["<core rule N>", "<standard chapter>", "<other AIR-NN>"]
```

The list AIR-01..AIR-14 is closed; new risks — only through a change to the full RENAR Standard.

Category ↔ NIST AI RMF trustworthiness characteristics (for verifiability of the claim "based on NIST AI RMF"):

category	NIST AI RMF trustworthiness characteristic
`hallucination` / `data-quality`	Valid & Reliable
`injection` / `adversarial`	Secure & Resilient
`drift`	Valid & Reliable; Safe
`bias`	Fair — with Harmful Bias Managed
`sgnl-failure`	Safe; Accountable & Transparent

`privacy`	Privacy-Enhanced

> ***ISO/IEC 23894:2023 references.*** *The AI-risk categories in 23894:2023 are located in* ***Annex A*** *(risk sources); Clause 6 describes the risk-management process (per ISO 31000). The "Annex A — …" descriptors in the register are risk-source labels; the exact mapping to Annex A items is subject to reconciliation when a formal claim is made.*

2. Register AIR-01..AIR-14

Metadata for all 14 risks (Sev=Severity, Like=Likelihood):

AIR	Name	Category	Sev	Like	ISO 23894 (Annex A)	NIST RMF	Status
01	Hallucination in AI-generated requirements	hallucination	**High**	High	Output reliability	Measure	active → mitigated at mature RENAR levels
02	Prompt injection via client TZ	injection	**High**	Low-Medium	Adversarial inputs	Manage	monitoring
03	Model drift / version change	drift	Medium	High	Model lifecycle	Manage	monitoring
04	Bias in AI requirements generation	bias	Medium	Medium	Fairness	Measure	active
05	Single-model failure (no diversity)	sgnl-failure	Medium	High	Single point of failure	Manage	mitigated with full pipeline
06	Test fitting / greening of tests	sgnl-failure	**High**	Medium	Verification integrity	Measure	mitigated with spot-check
07	Hallucinated citations	hallucination	Medium-High	Medium	Output reliability	Measure	monitoring
08	Adversarial inputs in clients data (runtime)	adversarial	**High**	Low	Adversarial inputs	Manage	out-of-scope (application-level)
09	Privacy leakage via AI logs	privacy	**High**	Medium	Privacy	Govern	active
10	Knowledge graph	data-quality	Medium	Low	Data integrity	Map	monitoring

	poisoning						
11	Reconciliation false-positive overload	sgnl-failure (process)	Low-Medium	Medium	Verification integrity	Manage	monitoring
12	Cost runaway (uncontrolled AI spend)	sgnl-failure (operational)	Medium	Medium	Cost governance	Manage	active
13	Stakeholder does not understand AI-generated requirements	data-quality (UX)	Medium	Medium	Transparency	Govern	active
14	Vendor lock-in to specific LLM provider	sgnl-failure (operational)	Medium	Low-Medium	Vendor risk	Govern	monitoring

Descriptions + impact + mitigations — below.

AIR-01. Hallucination in AI-generated requirements

When generating BR/SR/SPEC, an AI agent may "make up" statements that are not in ADAPT or the TZ. Impact: scope creep, dispute at acceptance, features that the client did not require. **Mitigations:** source citation (standard/04 §4.10.2 — every statement references ADAPT §N, and when no ADAPT was created (standard/07 §7.4.1) — directly a TZ section via `source.tz-section` + `source.adversarial-review-ref`); adversarial review (a critic model from a different vendor); Hallucination Rate metric ≤1% at mature RENAR levels.

AIR-02. Prompt injection via client TZ

A malicious client may embed hidden instructions for the AI into the TZ ("ignore previous instructions and ..."). Impact: data leakage, malicious changes to requirements, violation of the security policy. **Mitigations:** sandboxing of the AI agent on import (the model treats the TZ as passive data, stated explicitly in the system prompt); the input gateway checks for known injection patterns; a suspicious pattern → escalation, not auto-process.

AIR-03. Model drift / version change

Anthropic / OpenAI / Google update their models — the same prompt with the same TZ may give a different output six months later. Impact: inconsistency between requirements within a single project; inability to reproduce exactly the generation of an old artifact. **Mitigations:** model versioning in `ai-provenance.generated-by` (exact version + date); eval tests for SPEC-AI are run when the model changes; on regeneration — a diff against the old version and an assessment of the changes.

AIR-04. Bias in AI requirements generation

The model has training-data bias — when generating BR it may ignore stakeholders from particular groups (accessibility users, non-English locales, specific regulatory regimes). Impact: requirements do not cover the full spectrum of users; the product is discriminatory or non-compliant. **Mitigations:** multi-model

agreement for `priority=must` BR (different models — different biases); a stakeholder map is mandatory in BR (explicit enumeration); an adversarial critic with the prompt "check for missing stakeholders / accessibility considerations".

AIR-05. Single-model failure (no diversity)

If all artifacts are generated by a single model, its errors propagate systematically. It "hallucinates" a particular pattern — all requirements inherit it. Impact: systematic distortion of requirements across the project. **Mitigations:** multi-model for `priority=must` BR; an adversarial critic with a different model; isolation of the judge model from the production model in eval tests (SPEC-AI: `judge-model.vendor ≠ production-model.vendor`, see 02-schemas.md §6.2).

AIR-06. Test fitting / greening of tests

An AI agent has a trivial path to greening a failing test — weaken the Pass criterion. This passes code review because "the test is green." Impact: false confidence; defects pass into production. **Mitigations:** the `[test-spec-change]` marker is mandatory for changing Pass/Fail (a separate approval); spot-check of 5 passing TC once per iteration (standard/09 §9.14); the Test-spec Drift Rate metric (standard/12 §12.3).

AIR-07. Hallucinated citations

An AI agent writes the citation `[TZ-2026-001 §4 line 142]`, but in the real TZ §4 line 142 is about something else. The citation looks like evidence, but the evidence is false. Impact: source citation becomes a fiction; the trace chain breaks under audit. **Mitigations:** a citation validator hook (parses the citation, opens the referenced document, verifies conformance); pre-commit/pre-approval block on an invalid citation.

AIR-08. Adversarial inputs in clients data (runtime)

The client sends data (via forms, API) deliberately constructed to manipulate an AI component at runtime (not at the requirements-generation stage). Impact: similar to AIR-02, but in production runtime. **Mitigations:** input sanitization at the API gateway; constrained generation (structured outputs only); rate limiting per user. **Status: out-of-scope** — application-level security; RENAR requires SR-level coverage (SPEC-SEC threat model), but runtime protection is a task of implementation, not of normalizing requirements.

AIR-09. Privacy leakage via AI logs

When generating an artifact, an AI agent has PII in its context (from the TZ or an interview). Generation logs (tool events, audit records) may store this PII. Impact: PII ends up in logs, in `ai-provenance`, in training data (if used). **Mitigations:** PII redaction in prompts before sending to the LLM; `data-classification` tracking; disable training on conversations (Anthropic/OpenAI privacy settings, DPA); TTL on event logs with PII.

AIR-10. Knowledge graph poisoning

If the KG is used as primary search for AI agents, an incorrect edge can "poison" all subsequent AI queries that rely on that graph. Impact: the AI generates requirements based on wrong context, systematically.

Mitigations: the KG is derived from frontmatter, not edited directly (see 05-knowledge-graph-schema.md); CI validation of the graph on every change (no circular dependencies, no orphan approved); a reconciliation agent verifies integrity weekly.

AIR-11. Reconciliation false-positive overload

With overly sensitive rules, a reconciliation agent generates many false-positive findings. The Architect starts ignoring them → real findings drown. Impact: reconciliation loses value, the discipline does not scale. **Mitigations:** tunable thresholds in the project configuration; a Reconciliation Findings/Week metric (if it grows without real issues — re-calibration); the Architect may reject findings with a rationale — feedback for tuning the agent's prompt.

AIR-12. Cost runaway (uncontrolled AI spend)

Without budget tracking, AI generation (especially with multi-model, adversarial, eval) may consume tokens disproportionately to project size. Impact: financial losses; the practice becomes unprofitable. **Mitigations:** an `ai-budget` field in frontmatter (target + actual); an aggregated cost metric at project level; a cap per project; an alarm when approached; a recommendation engine "Sonnet/Haiku for routine work, Opus only for `priority=must` BR".

AIR-13. Stakeholder does not understand AI-generated requirements

The AI generates SR in a technical style; the client / a non-technical stakeholder does not understand it during review → approval becomes a formality. Impact: QG-ADAPT-approve / QG-4 acceptance loses meaning; the dispute rate at acceptance grows. **Mitigations:** the style guide (04-ai-style-guide.md); BR in business language (technologies — in SPEC-*, not in BR); a human-readable summary in every BR/SR — a short section, understandable without a technical background.

AIR-14. Vendor lock-in to specific LLM provider

All prompts are optimized for a specific provider (Anthropic Claude). If the provider changes pricing/availability — migration requires rewriting all prompts. Impact: operational risk, costs, business continuity. **Mitigations:** provider-agnostic prompts where possible (avoid vendor-specific tool syntax); multi-model agreement for `priority=must` is normative at RENAR-5 (standard/11 §11.8.1) — guarantees a second provider in the pipeline; periodic test runs on a backup provider.

---

3. Risk matrix

```
Severity / Likelihood
         │  Low     Medium                      High
─────────┼──────────────────────────────────────────────
High     │  AIR-08  AIR-02, 06, 07, 09          AIR-01
Medium   │  AIR-10  AIR-04, 11, 12, 13, 14      AIR-03, 05
Low      │  —       —                           —
```

Range values from §2 (`Medium-High` , `Low-Medium`) are placed at their upper bound. Critical and High risks in the top-right quadrant are the mitigation priority.

4. Mitigation matrix

Which mitigations cover which risks (compensating mechanisms: a single mitigation is rarely sufficient; high risks require ≥ 2 independent mechanisms):

Mitigation	Covers risks
Source citation (standard/04 §4.10.2)	AIR-01, AIR-07
Adversarial review (a different model)	AIR-01, AIR-04, AIR-05
Spot-check of passing TC (standard/09 §9.14)	AIR-06
Multi-model for `priority=must`	AIR-04, AIR-05, AIR-14
Judge isolation in SPEC-AI	AIR-05
AI provenance (model + version + date)	AIR-03, AIR-09
Citation validator hook	AIR-07
Input sandbox / sanitization	AIR-02, AIR-08
PII redaction + DPA	AIR-09
KG validation in CI	AIR-10
Reconciliation tunable thresholds	AIR-11
`ai-budget` field + project cap	AIR-12
Style guide + business-language BR	AIR-13

5. Operational governance

Review cadence. Monthly: AIR-01, AIR-02, AIR-06, AIR-07, AIR-09 (high-impact runtime risks). Quarterly: the rest. On-incident: any risk in which an incident occurred → root cause → mitigation.

Owner. Default — the project Architect. For AI-specific risks — the AI Governance Lead (if one exists in the organization).

Storage. The project's risk register is a separate artifact:

```
<project>.req/
  governance/
    ai-risk-register.md          # snapshot of this reference + project-specific
notes
    review-log.md                # review history with dates and signatures
```

On a substrate that does not support directories — an equivalent namespacing.

Reconciliation agent. A weekly run updates the `status` and `last-reviewed` fields of each AIR. If a status changes (e.g., monitoring → active) — an alert to the Architect.

6. Relationship to the standard

AIR	RENAR Core / Standard
AIR-01, 07	Standard ch.4 §4.10.2 (source citation) + ch.7 §7.4.1 (reactive ADAPT: `source.adapt` or `source.tz-section` + `source.adversarial-review-ref`)
AIR-04, 13	Standard ch.5 (roles) + style guide §04
AIR-05	Standard ch.9 §9.6.2 (judge ≠ production isolation) + SPEC-AI
AIR-06	Standard ch.9 §9.7 (pos/neg pairing) + §9.13 (protection from test fitting) + §9.14 (spot-check) + §9.18 (test authorship isolation and red history) + QG-2
AIR-09	Standard ch.4 §4.10.1 (`ai-provenance`) + SPEC-SEC
AIR-12	Standard ch.12 §12.3.9 (Cost per Approved Requirement) + ch.11 §11.8.1 (cost/latency budget)

7. What the risk register does NOT cover

This register focuses on AI-specific risks of the RENAR process. **It does not replace:** the organization's general security risk register (ISO 27001); the compliance risk register (06-compliance.md); the application-level threat model of a specific project (SPEC-SEC). Regulator-mandated requirements (AI Act high-risk, FZ-152) — separate artifacts; this register is the operational tier.

AI Risk Register RENAR 1.0 — renar.tech

AI Style Guide — generating RENAR artifacts

> ***Purpose:*** *style, tone, structure, length, and lexicon for AI agents generating RENAR artifacts (ADAPT, BR, SR, SPEC, TC, Impact Analysis). It eliminates a class of risks — "different models → different style → drift in perception" (see AIR-04, AIR-13 in 03-ai-risk-register.md). It complements the reference/06 EN Style Guide for human editors.*

Context: the AI agent in RENAR is the **regular primary author** of artifacts (standard/00 §0.2.1); the AI-vs-Architect distribution of roles — standard/05 (§5.2.1, §5.3.2, §5.6 RACI). This document normalizes style for the regular mode, not for exceptions.

Industry analogues: Google Developer Documentation Style Guide, Microsoft Style Guide for technical writing, Diátaxis. The goal is a single, even tone regardless of the model or prompt engineer.

---

1. Principles

1.1 Style = contractual precision

A RENAR artifact is a **contractual document** between the client, the team, and the system. No metaphors, no "colorful" language, no associations. Only unambiguous statements.

Good:

> *The system MUST accept a registration request via POST /auth/register.*

Bad:

> *Users will be able to easily and conveniently create accounts via a modern API.*

1.2 One artifact = one idea

Each BR / SR / SPEC / TC covers **one** concept. If the AI agent is inclined to "pack" 3 topics into one SR, it MUST decompose rather than pack.

Packing signal: the conjunction "and" in the title, multiple actions in a single statement.

Bad:

> *SR-05: Registration, authentication, and password recovery*

Good:

> *SR-05: Client registration SR-06: Client authentication SR-07: Password recovery*

1.3 No claim to completeness

The AI MUST NOT "augment" a requirement on its own if the source (ADAPT or TZ) does not contain it. Every statement either has a citation or is marked `derived` with explicit justification.

Bad:

```
- Email uniqueness is checked at registration.
- Phone uniqueness is checked at registration.   ← not in ADAPT, the AI made it
up
```

Good:

```
- Email uniqueness is checked at registration. [ADAPT-001 §14.1 Forward]
```

If the AI believes a phone check is needed, it does not write it into the SR; instead it adds it to the Critic output: "a phone uniqueness check may be needed — not in ADAPT, requires a backward finding to the Stakeholder."

1.4 Unambiguity over beauty

When choosing between a precise long sentence and a compact ambiguous one, the AI chooses the precise long one. Length is not the enemy, ambiguity is the enemy.

---

2. Structure and length by artifact type

2.1 ADAPT

Body: 200-800 lines (depends on the size of the TZ).

Mandatory sections:

- `## Summary` — 3-5 paragraphs for a one-page read by the Architect and the adversarial reviewer (standard/07 §7.8.2).
- `## Term mapping` — a "client → engineer" table.
- `## Forward: interpretation by TZ section` — a section for each § of the TZ.
- `## Backward: discovered problems` — B-NNN entries with a lifecycle.
- `## Backward findings digest` — a table by category.
- `## Generated artifacts` (auto after approval).
- `## ADAPT change history` — substrate-native, auto-generated (standard/07 §7.8.2).

Tone: engineering throughout. ADAPT is an internal artifact of interpretation; its audience is the engineer, the AI agent, the adversarial reviewer, and the verifier — the client never sees it (standard/07 §7.5). Client-facing wording lives in ACTZ (standard/07 §7.13), where the Architect escalates backward findings.

2.2 BR — Business Requirement

Body: 30-80 lines.

Mandatory sections:

- `## Need` — one sentence following the template `[Role] MUST [action] in order to [business goal].`

- `## Success criteria` — 3-7 measurable items.
- `## Context` — 5-15 lines.

Prohibited:

- Technologies ("via REST API", "Postgres", "React").
- Specific screens or DB fields.
- Author's musings ("we think", "perhaps", "it would be nice").

Tone: formal, imperative ("MUST", not "MAY").

2.3 SR — System Requirement

Body: 40-150 lines.

Mandatory sections:

- `## Requirement` — one sentence following the template `The system MUST [behavior]. [Condition].`
- `## Behavior` — 5-20 items with inline citations to ADAPT.
- `## Constraints` — 3-10 items; mandatory if applicable.
- `## Link to SPEC` — mandatory if `constrained-by[]` is present (standard/06 §6.6.3).

Optional sections (where applicable):

- `## Not part of this requirement` — explicit out-of-scope.
- `## Related SR` — cross-links.

Prohibited:

- DB table names, specific functions, classes.
- Frameworks ("via FastAPI", "React component").
- Specific UI paths ("button in the top-right corner") — that belongs in SPEC-UI.

Tone: formal, precise, behavioral ("the system responds with 422", not "an error is returned").

2.4 SPEC-* (11 types)

Body: 80-400 lines (varies by type).

Common mandatory sections (see 02-schemas.md §5):

- `## Purpose`
- `## Scope` (in / out)
- `## <Type-specific sections>`
- `## Link to requirements`
- `## Link to other SPEC`
- `## Verification`
- `## Open questions`

Tone by type:

- **SPEC-ARCH / API / DATA / INT / SEC / OPS:** technical-analytical. Specific technologies are allowed.

- **SPEC-UI:** user-centric narrative with behavioral precision. "The manager sees the order feed, taps an order, and a detail screen opens", not "the UI must have a listing component".
- **SPEC-AI:** model card style — capabilities, limits, failure modes, eval criteria.
- **SPEC-PROC:** procedural narrative with BPMN / state machine references.

2.5 TC — Test Case

Body: 30-80 lines.

Mandatory sections:

- `## Context` — which item of the verified artifact the TC references.
- `## Preconditions` (Given) — 2-5 lines.
- `## Steps` (When / action) — 1-3 lines.
- `## Pass criterion` (Then / Pass) — a binary, observable, reproducible criterion.
- `## Fail criterion` — a list of observable signs of violation, **including** possible side effects (security leaks, missing audit logs).
- `## Postconditions` — the expected state after the run.
- `## Out of scope` — what is deliberately not checked, with a pointer to the covering TC.

> *The criterion section names (`## Pass criterion` / `## Fail criterion`) are canonical machine-detectable headings (standard/09 §9.4), detected by the enforcement hook (standard/10 §10.11.3); do not rename them.*

Tone: executable, crisp ("X is performed", "Y is returned").

2.6 Impact Analysis

Body: 50-200 lines (depends on the size of the delta).

Mandatory sections:

- `## Affected requirements` — a table with the change type.
- `## Affected SPEC` — a table with the action.
- `## Affected TC` — a table with the action.
- `## Affected backlog tasks`.
- `## Open questions` — what needs to be clarified with the Stakeholder via a backward in the delta-ADAPT.

Tone: analytical, without judgments. Not "this is a bad change", but "this change affects X requirements and Y tasks".

---

3. Lexicon

3.1 Use (canonical)

- "The system MUST ..." (modality).
- "MUST" / "MUST NOT" (RFC 2119 vocabulary).
- "returns" (HTTP/API behavior).

- "preconditions" / "postconditions" (formal testing terms).
- "requirement" / "statement" / "assertion".
- "acceptance criterion".
- "backward finding" / "forward interpretation" (RENAR canonical, from ADAPT).

3.2 Prohibited

Word/phrase	Replace with	Why
"We'd like it to"	"The system MUST"	Modality must be strict.
"Convenient"	a specific measurable criterion	Subjective.
"Modern"	specific tech requirements in SR/SPEC	Marketing language.
"High-quality"	measurable quality characteristics from ISO/IEC 25010	Subjective.
"Fast"	p95/p99 latency with a number	Vague.
"Reliable"	uptime / reliability with a number	Vague.
"Maybe" / "Perhaps"	(never in requirements)	Hedging.
"Most likely"	(never)	Same.
"Preferably"	`priority: should` / `could` in the frontmatter	Mixing channels.
"Intuitive"	criteria for clarity (Flesch reading ease, etc.)	Subjective.
"Smooth experience"	UX criteria in SPEC-UI	Marketing.
"Seamless"	a specific technical criterion	Marketing.
"Support" (without explaining how)	a specific SR-level criterion	Vague.
"Integrate with X" (without a contract)	SPEC-INT with a counterparty	Vague.

3.3 Use Cases vs Stories vs Requirements

RENAR uses **only the canonical terminology** from 01-glossary.md. The AI agent **is not permitted** to call a requirement a "User Story", "Use Case", "Scenario", or "Capability" — these are prohibited synonyms. The full list of prohibited terms and substitutions — in 01-glossary.md §4.

4. Statement templates

4.1 BR statement

Template: `[Role]` MUST `[action]` in order to `[business goal]`.

Example:

> *The user MUST automatically receive and store notifications from selected applications in the background, in order not to miss important messages.*

4.2 SR statement

Template: `<actor>` `<action>` `<condition>` . `<consequence>` .

Example:

> *On first launch the system checks for the NotificationListenerService permission. If the permission is not granted, an onboarding screen with a "Grant access" button is displayed.*

4.3 TC Pass criterion

Template: `<actor>` `<action>` `<input>` . `<observable>` `<measurement>` .

Example:

> *POST /auth/login with body `{"email":"x@y.z", "password":"correct"}` . Response status = 200, body contains a JWT with `exp = now + 24h ± 1m` .*

4.4 TC Fail criterion

A list of observable signs of violation, **including** possible side effects:

```
- Response status ≠ 200 (authorization error).
- The 401 response names the specific field that is incorrect (information leak).
- A plaintext password is written to the system log (security violation).
- A login-notification email is NOT sent on successful authorization (missing
side effect).
```

Not allowed: "Pass is not met". This is not a Fail criterion, it is a negation.

---

5. Citation conventions

5.1 Inline citation

Every statement in a BR/SR/SPEC has an inline citation in square brackets:

> *On first launch the onboarding screen is displayed. [ADAPT-001 §14.1 Forward]*

Format: `[<id> <section>]` or `[<id> <section> line <N>]` for a precise line.

5.2 Derived marker

If a statement is not a quote from ADAPT but logically inferred:

> *The `Back` button is disabled on the onboarding screen. [ADAPT-001 §14.1 Forward, derived: ADAPT requires "onboarding cannot be skipped" → blocking the system `back`]*

The `derived` explanation is mandatory.

5.3 Multi-source

If a statement is supported by several sources:

> *The password is stored encrypted. [ADAPT-001 §4 NFR-002, ISO 27001 A.10.1.1]*

5.4 What is NOT a citation

- A reference to the code itself ("see `auth/login.py` ") — not a requirement source.
- A reference to a ticket in a bug tracker — not a requirement source (see 01-glossary.md §1 Authority chain).
- A reference to a chat / verbal conversation without an entry in ADAPT backward → asked-to-client → answered.

---

6. System prompt for the AI generator

Every AI agent generating a RENAR artifact receives a system prompt composed of:

1. **Role:** "You are a requirements architect under the RENAR standard."
2. **Style guide reference:** a link to this document.
3. **Glossary reference:** a link to 01-glossary.md.
4. **Constraints:**
   - Every statement is either with a citation or `derived` with an explanation.
   - Use only canonical terms.
   - Structure and length — per §2 of this document.
   - No prohibited words from §3.2.
5. **Output schema:** the exact YAML frontmatter format (see 02-schemas.md).
6. **Examples:** 2-3 good/bad examples for each artifact type.

Prompt templates are stored in the organization's `prompts/` directory with versioning:

- `prompts/adapt-from-tz.md@v2.1`
- `prompts/decompose-adapt.md@v2.1`
- `prompts/generate-tc-pos-neg.md@v1.0`
- `prompts/critic-review-sr.md@v1.2`

The artifact's `ai-provenance.prompt-template` frontmatter field holds the exact version.

---

7. Style for different models

Different LLMs have different tendencies. The style guide accounts for this through prompt constraints:

Model	Tendency	Mitigation
Claude Opus	Verbose, prone to hedging ("maybe", "perhaps")	Explicit constraint in the prompt: "no hedging modal verbs".
Claude Sonnet	Concrete, may miss an edge case	The adversarial critic catches gaps.
GPT-4 / o-series	Marketing-language tendency	Explicit blacklist of words from §3.2.
Mini / Haiku	Hallucinated citations	Citation validator hook (see AIR-07).
Gemini Pro	Sometimes mixes RU/EN	Lang constraint in the prompt + post-validation.

The style guide does not prohibit the use of any model, but it requires enforcement via post-generation validation.

8. Validation pipeline

After a RENAR artifact is generated, automatic validation runs:

```
AI generates artifact
    ↓
Style validator (a separate AI with a different prompt, or rule-based)
    ├── citation check (each assertion has [...] or a derived marker)
    ├── lexicon check (no forbidden words from §3.2)
    ├── structure check (required sections present)
    ├── length check (within bounds from §2)
    ├── canonical terms check (no forbidden synonyms)
    ├── modal verb check (no hedging)
    ↓
On fail: regenerate with a refined prompt, or human review.
On pass: enter the draft → ready transition (then QG-1 / adversarial review /
approval).
```

The validator hooks are native to the substrate. On git — pre-commit; on a document store — pre-save document validator; on any other — an equivalent gate.

9. Stylistic decisions for RU/EN

9.1 Body language

The body of an artifact is written in the project's language:

- Russian client → RU.
- International → EN.
- Bilingual → primary lang in the `lang:` frontmatter field; the second lang — a separate artifact with `replaces` / `replaced-by` links, or a translations subfolder.

9.2 Frontmatter is always canonical

Frontmatter fields are always canonical (English latin):

- `type: BR` (not `тип: БТ`).
- `status: approved` (not `статус: утверждено`).
- `priority: must` (not `приоритет: обязательно`).

The UI displays Russian translations (see 01-glossary.md §3.5). Frontmatter is for machines, not for the UI.

9.3 Mixed-lang is prohibited

Within a single statement, mixing RU+EN is not allowed:

Bad:

> *Система должна возвращать `JWT token` после successful login.*

Good (RU):

> *Система должна возвращать JWT-токен после успешной аутентификации.*

Good (EN):

> *The system must return a JWT token after successful authentication.*

Technical terms (`JWT`, `OAuth`, `gRPC`, `RBAC`) are allowed in any language without translation.

---

10. Cross-references

- The closed list of canonical terms and prohibited synonyms — 01-glossary.md.
- Formal frontmatter schemas — 02-schemas.md.
- AI risks and mitigations — 03-ai-risk-register.md.
- Knowledge graph schema (used by the validator for cross-reference checks) — 05-knowledge-graph-schema.md.

AI Style Guide RENAR 1.0 — renar.tech

Knowledge Graph Schema

> ***Purpose:*** *the formal knowledge-graph (KG) schema for RENAR projects — nodes, edges, properties, query patterns. The KG is a* ***derived view*** *of RENAR artifacts for semantic queries by AI agents and for reconciliation. Machine-readable edge types — §3; the normative link fields — standard/06 §6.10, standard/08.*

The KG is **not the Source of Truth**. The Source of Truth is the artifacts (ADAPT / BR / SR / SPEC / TC) on the substrate. The graph is derived from frontmatter and is never edited directly. If the graph contradicts the artifacts → rebuild the graph, do not edit the artifacts.

1. Use cases

Without the KG, an AI agent has only a flat search (FTS5/RAG + `parent` / `children` direct hops + keyword grep) — a syntactic context. The graph adds a semantic one:

Query	Without the graph	With the graph
All BR affecting the Sales Cycle KPI	Grep + parse frontmatter	A single Cypher query
When SR-05 changes — which tasks and SPEC are affected	Scan all references	Single graph traversal
Which SPEC-AI require high-risk classification under the AI Act	Manually	One query
Stakeholder map for a project	Several queries	Single subgraph extraction
Cross-project dependencies via SPEC-INT	Federation API calls	Federated graph query
Whether requirement SR-12 has started to degrade	Manual analysis	Trend analytic on node properties
Stale TC (verifies an SR whose version was updated, last-run on the old one)	Manual check	One query

2. Node types (closed list)

Type	Source	Identity
`Requirement`	BR / SR / TR artifacts	`<id>`
`Specification`	SPEC-* artifacts (11 types)	`<spec-id>`
`ADAPT`	ADAPT-NNN artifacts	`<adapt-id>`

`BackwardFinding`	B-NNN entries inside an ADAPT	`<adapt-id>:B-NNN`
`TestCase`	TC artifacts (incl. `tc-type: contract`)	`<tc-id>`
`WorkOrder`	TZ and delta-TZ	`<tz-id>`
`Stakeholder`	frontmatter `business-context.stakeholder`	`<stakeholder-id>`
`BusinessGoal`	frontmatter `business-context.business-goal` (deduplicated)	`<goal-id>`
`KPI`	frontmatter `business-outcome.kpi-name`	`<kpi-id>`
`Task`	TR / runtime task store	`<task-id>`
`CodeArtifact`	TC `automation.location`, code commits	`<repo>:<path>:<symbol>`
`Decision`	Architectural decision records	`<decision-id>`
`DeadEnd`	Failed approaches (optional)	`<deadend-id>`
`RiskItem`	AI Risk Register entry	`AIR-NN`
`Compliance`	Compliance standard reference	`<std>:<control>`
`Template`	Requirements library template	`<template-id>`

The list is closed; new node types — only via an amendment to the full RENAR Standard.

3. Edge types (closed list)

3.1 Hierarchy and derivation

Edge	From	To	Semantics
`parent`	Requirement	Requirement	A.parent = B → B is parent of A (BR → SR → TR)
`derived-from-adapt`	Requirement / Specification	ADAPT	Artifact derived from an approved ADAPT section
`derived-from-template`	Requirement / Specification	Template	Template (with a version pin)
`replaces`	Requirement / Specification	Requirement / Specification	new replaces old (deprecated)
`supersedes`	Requirement / Specification	Requirement / Specification	strictly newer version (post-delta)
`parent-adapt`	ADAPT	ADAPT	delta-ADAPT → main ADAPT

3.2 ADAPT specifics

Edge	From	To	Semantics
`from-tz`	ADAPT	WorkOrder	source-tz pointer
`parent-adapt`	ADAPT	ADAPT	delta-ADAPT → root (`parent-adapt`)
`supersedes`	ADAPT	ADAPT	superseding ADAPT → superseded one; inverse `superseded-by` (standard/07 §7.6.4); the target moves to `superseded`
`contains-backward`	ADAPT	BackwardFinding	B-NNN entry
`backward-asks-stakeholder`	BackwardFinding	Stakeholder	who answers
`decided-in`	BackwardFinding	ACTZ	Pointer to the clause of a signed ACTZ that records the client's decision (standard/07 §7.4.5); a pointer to an unsigned ACTZ is fatal

3.3 SPEC graph (parallel axis)

Edge	From	To	Semantics
`constrained-by`	Requirement	Specification	SR constrained by SPEC-* (typed edge)
`implements-spec`	Task	Specification	TR implements SPEC-*
`depends-on`	Specification	Specification	SPEC depends on another SPEC (DAG)
`referenced-by`	Specification	Requirement / Task	inverse (auto-derived)

3.4 Verification and implementation

Edge	From	To	Semantics
`verifies`	TestCase	Requirement / Specification	TC verifies artifact
`verified-by`	Requirement / Specification	TestCase	inverse (auto-derived)
`implements`	Task	Requirement	Task implements requirement
`realises-in`	TestCase	CodeArtifact	TC.automation.location
`linked-defect`	Requirement	Task (defect type)	Bug on this requirement

3.5 Provenance

Edge	From	To	Semantics
`from-order`	ADAPT / Requirement	WorkOrder	source.tz-section reference
`delta-from`	WorkOrder	WorkOrder	delta-TZ → base TZ
`cited-in`	Requirement / Specification	WorkOrder	inline citation pointer to a TZ section
`decided`	Requirement / Specification	Decision	Decision during decomposition
`deadend`	Requirement / Specification	DeadEnd	Failed approach

3.6 Business / governance

Edge	From	To	Semantics
`owned-by`	Requirement	Stakeholder	business-context.stakeholder
`goal`	Requirement	BusinessGoal	business-context.business-goal
`impacts-kpi`	BusinessGoal	KPI	KPI driven by goal
`compliance-with`	Requirement / Specification	Compliance	compliance entry
`risk-mitigates`	Requirement / Specification	RiskItem	mitigates AIR-NN
`risk-introduces`	Requirement / Specification	RiskItem	introduces new risk (warning trigger)

3.7 Cross-project / integration

Edge	From	To	Semantics
`participates-in`	Specification (SPEC-INT)	Specification (SPEC-INT)	SPEC-INT participants — the parties to the integration contract
`cross-deps-on`	Task (project A)	Task (project B)	Cross-project dependency
`integrates-via`	Requirement	Specification (SPEC-INT)	Via a SPEC-INT contract

3.8 Edge properties

Edges carry properties (Cypher-style):

```
(TC:TestCase)-[v:verifies {requirement-version: "1.2", confidence: "high"}]->
(SR:Requirement)
(BR:Requirement)-[c:compliance-with {control: "A.5.34", rationale: "PII
protection"}]->(GDPR:Compliance)
(WO:WorkOrder)-[d:delta-from {effective-date: "2026-05-15"}]->(WO-prev:WorkOrder)
(SR:Requirement)-[cb:constrained-by {since-version: "1.0"}]->(SPEC:Specification)
```

4. Cypher-style query examples

4.4 Stale TC (criteria-version drift)

```
MATCH (r:Requirement)<-[:verifies]-(tc:TestCase)
WHERE tc.last-run.requirement-version < r.version
RETURN r.id, tc.id, tc.last-run.requirement-version, r.version
```

4.5 Multi-hop: code → test → SR → BR → KPI

```
MATCH (code:CodeArtifact {path:"src/auth/registration.py"})
      <-[:realises-in]-(tc:TestCase)
      -[:verifies]->(sr:Requirement {type:"SR"})
      -[:parent*1..2]->(br:Requirement {type:"BR"})
      -[:goal]->(g:BusinessGoal)
      -[:impacts-kpi]->(k:KPI)
RETURN code.path, sr.id, br.id, g.name, k.name
```

"Which KPI depend on this code file?" — in a single query.

4.6 SPEC dependency cycle detection

```
MATCH path=(s1:Specification)-[:depends-on*]->(s1)
RETURN path
```

Returning rows → a violation of the DAG invariant (02-schemas.md §9).

4.7 ADAPT with open backward findings

```
MATCH (a:ADAPT {status:"draft"})-[:contains-backward]->(b:BackwardFinding
{status:"open"})
RETURN a.id, count(b) as open_count
ORDER BY open_count DESC
```

> ***Note on numbering.*** *§4.4-§4.7 are used to preserve the cross-refs from 02-schemas.md §9 to specific queries (Stale TC, SPEC cycle, ADAPT open). Additional queries (BR→KPI, SR-affected tasks, PII→GDPR scope) are derived from the patterns in §4.4-§4.7 plus the node/edge tables in §2-§3.*

5. Derivation rules

The KG is derived from artifact frontmatter. "Direct editing of the graph" does not exist.

Node type	Source in frontmatter		Edge	Source
`Requirement`	`id`, `type ∈ {BR,SR,TR}`		`parent`	`parent.id` field
`Specification`	`id`, `type ∈ {SPEC-ARCH..SPEC-DOC}` (all 11 types, standard/08 §8.3)		`verifies`	`verifies[].id` in a TC
`ADAPT`	`id`, `type: ADAPT`		`constrained-by`	`constrained-by[]` in an SR
`BackwardFinding`	ADAPT body parsing		`depends-on`	`depends-on[]` in a SPEC
`TestCase`	`id`, `type: TC`; derived: `last-run.*`, `last_modified` (§7 SQLite schema), `criteria_changed_at` (§6.5)		`implements-spec`	`implements-spec[]` in a TR
`WorkOrder`	TZ-NNN frontmatter		`owned-by`	`business-context.stakeholder` → Stakeholder
`Stakeholder` / `BusinessGoal` / `KPI`	deduplicated from `business-context.*` / `business-outcome.kpi-name`		`compliance-with`	`compliance[]` array
`Compliance`	`compliance.standard` + `compliance.control`		`derived-from-adapt` / `from-order` / `delta-from`	`source.adapt` / `source.tz-section` / `parent-adapt`

Refresh policy. The graph is **rebuilt** on a change in the `.req` repository / collection (post-commit/post-save hook), on import of TC last-run from the CI bot, and on creation/update of a task. The rebuild is **incremental** (only the affected nodes and edges); a full rebuild — once a week by the reconciliation agent.

6. Validation queries (reconciliation)

6.1 Orphan approved requirements

```
MATCH (r:Requirement {status:"approved"})
WHERE NOT (r)-[:verified-by]->()
RETURN r.id  // approved without a TC — warning
```

6.3 Circular parent chain

```
MATCH path=(r1:Requirement)-[:parent*]->(r1)
RETURN path
```

6.5 Test fitting suspicious (AIR-06 signal)

```
MATCH (tc:TestCase)
WHERE tc.last_modified < tc.criteria_changed_at
  AND tc.last-run.result = "pass"
RETURN tc.id  // criteria changed recently, yet passing right away — suspicious
```

> *Properties of a `TestCase` node: `last_modified` — from the node table (see §7 SQLite schema); `criteria_changed_at` — derived: the timestamp of the last change-record carrying the `[test-spec-change]` marker (01-glossary.md §2.12). Both are populated during graph derivation (§5).*

6.6 SR without constrained-by (missing SPEC links)

```
MATCH (sr:Requirement {type:"SR", status:"approved"})
WHERE NOT (sr)-[:constrained-by]->(:Specification)
RETURN sr.id  // SR approved but not linked to any SPEC — warning
```

> ***Additional validation queries*** *(broken citations, orphan stakeholders, orphan SPEC) — patterns derived from §6.1 + §6.5 plus the node/edge tables in §2-§3.*

7. Substrate-native implementations

The graph is derived from the frontmatter of all `.md` files in `<project>.req/` plus the task store. The recommended implementation for a git substrate is an embedded SQLite with tables `nodes(id, type, properties JSON, last_modified)` and `edges(from_id, to_id, edge_type, properties JSON)`, indexed on `from_id`, `to_id`, `edge_type`. An alternative for large projects: an embedded graph DB (Kuzu, RedisGraph local).

Document substrates use native graph queries via design views / Cypher-like languages — no separate infrastructure is required.

Schema invariants (substrate-independent): node types and edge types are fixed (closed list). Properties MAY evolve (minor schema bump). Removing a node/edge type — a major schema bump + migration.

8. Federated queries (cross-project)

Coordinating multiple projects via KG federation. Example: "which cross-team integrations have version drift."

```
MATCH (s1:Specification {project:"auth", type:"SPEC-INT"})
      -[:participates-in]->(int:Specification)
      <-[:participates-in]-(s2:Specification {project:"billing"})
WHERE int.contract-version <> s1.implemented-version
RETURN s1.id, s2.id, int.id, int.contract-version, s1.implemented-version
```

Federation is a substrate-dependent operation; it is implemented via a convention (a cross-substrate query layer) or a native multi-tenant graph.

9. Implementation roadmap

Maturity level	Graph coverage
RENAR-1 / Core	KG optional; parsing frontmatter is sufficient.
RENAR-2 / Foundation	Basic graph: Requirement, TestCase, WorkOrder, Task; edges parent, verifies, verified-by, implements. Simple pre-built queries.
RENAR-3 / Team	+ SPEC, ADAPT, BackwardFinding, Stakeholder, BusinessGoal, KPI, Decision. Edges: constrained-by, depends-on, derived-from-adapt, owned-by, goal, impacts-kpi. AI prompts with graph context.
RENAR-4 / Enterprise	Full schema + reconciliation queries + federation. Visualization in the UI.
RENAR-5	Trend analytics, cross-substrate federation, embedded graph DB for large repos.

10. Cross-references

- Canonical termini for nodes and edges — 01-glossary.md.
- Formal frontmatter schemas (the derivation source) — 02-schemas.md.
- AI Risk Register (AIR-10 KG poisoning) — 03-ai-risk-register.md.
- Style guide (the validator uses the KG for cross-reference checks) — 04-ai-style-guide.md.

Knowledge Graph Schema RENAR 1.0 — renar.tech

RENAR EN Style Guide

> ***Normative artifact.*** *This document is the canonical Style Guide for the EN normative corpus of the RENAR Standard: terminology (§1), RFC-2119 EN wording (§2), formatting (§3).*
>
> ***Application:*** *any editorial change to EN content under* `standard/en/` , `reference/en/` , `core/en/` *(and, with soft application,* `guide/en/` *).*
>
> ***Relationship to the RU Style Guide.*** *This guide is a mirror of* `reference/06-ru-style-guide.md` *with* ***inverted RFC-2119 polarity****: EN canonical normative wording is UPPERCASE (* `MUST` */* `SHOULD` */* `MAY` *), and the RU lowercase carve-out (RFC 8174) does* ***not*** *apply to the EN corpus. Terminology that the RU corpus renders in Russian prose (RU Style Guide §1.15) is rendered here in native English.*
>
> ***Operational enforcement:*** `scripts/style-guide-check.js` *and companion gates. RU-targeted gates (* `check-substrate-term.js` , `check-process-vocab.js` *, the RU-prose checks inside* `style-guide-check.js` *) skip* `lang: en` *files; EN files are checked for structural integrity (fence-lang, links, section-refs, parity). The Style Guide sets the* ***rules****; the scripts perform* ***enforcement****.*
>
> ***Version history:*** *CHANGELOG.md.*

§0 Introduction

§0.1 Purpose

The Style Guide fixes the **form** of EN normative content:

- which terms stay latin (canonical identifiers, shared with the RU corpus), which are rendered in native English prose, and which loanwords are accepted (§1);
- the regime for RFC-2119 keywords (EN UPPERCASE canonical, per RFC 2119 and RFC 8174) (§2);
- canonical formatting of citations / code-fences / headings / tables / typography / frontmatter (§3).

The Style Guide does **not** dictate per-sentence wording — it sets constraints within which the editor applies judgement.

§0.2 Normative status

The Style Guide is a **normative companion** to the RENAR Standard.

- Any editorial change to EN normative content MUST conform to §1–§3.
- On a terminology conflict, `standard/04-terms.md` (canonical terminology) wins; §1 of this guide reflects that result.

§0.3 Scope

In scope: EN prose, headings, tables, lists, code-fences, and frontmatter under `standard/en/`, `reference/en/`, `core/en/`, `guide/en/`; cross-file links; normative semantics (no `MUST` → `SHOULD` downgrade); EN typography.

Out of scope: the RU corpus (`standard/`, `reference/`, `core/`, `guide/` proper — governed by the RU Style Guide); code under `scripts/`, `site/`, `.tausik/` (governed by tech-stack conventions); imagery / SVG (deferred); `research/` drafts pre-publish (author discretion).

EN directory convention (decision #80, task `en-dir-frontmatter`): each EN file mirrors its RU original at `<section>/en/<name>.md` with the **same file name** — e.g. `standard/en/00-introduction.md` mirrors `standard/00-introduction.md`, `reference/en/01-glossary.md` mirrors `reference/01-glossary.md`. The alternative `en/<section>/...` (a duplicated tree at the repository root) is rejected: it would break relative cross-link depth and per-section site navigation. Structural EN↔RU parity is enforced by `scripts/check-en-parity.js`.

§0.4 Hierarchy of authority

On a source conflict the first match wins:

1. `standard/04-terms.md` — canonical RENAR terminology.
2. `standard/01-scope.md §1.7` — closed-list policy.
3. EN Style Guide §1–§3 — operational normative form for the EN corpus.
4. RU Style Guide `reference/06` — for shared structural conventions (formatting, headings, code-fences) that are language-agnostic.
5. SENAR (the parent standard) — for general engineering terminology.
6. ISO/IEC/IEEE 29148:2018 — for requirements-engineering terms.

§0.5 Change procedure

Style Guide changes follow a formal procedure (mirrors the closed-list policy `standard/01 §1.7`):

1. Propose — `research/en-style-amendment-NNN-<topic>.md` with rationale.
2. Review — Style Guide editor (or project owner).
3. Pilot — an accepted amendment is pilot-validated on one chapter.
4. Lock-in — merged, with a version bump.
5. Retroactive — already-translated chapters are re-scanned for compliance.

§0.6 Versioning

Version	Status	Date
v1.0	**draft**	**2026-06-06 (EN translation epic — foundation)**

§1 Terminology

§1.1 Principles

Six normative principles govern EN terminology:

1. **No mass find-replace** — each term decision is contextual. Blind search-replace breaks normative meaning. The translator works per-occurrence.
2. **Preserved normative semantic** — a change that alters the RFC-2119 level (`MUST` / `SHOULD` / `MAY` discrimination) is a content change, not editorial. See §2.5.
3. **Reverse-calque guard** — a translator working from RU MUST NOT produce Russian-flavored English (literal calques of RU syntax, transliterated Russian terms, or word-order artifacts). Native idiomatic English is required. Conversely, the translator MUST NOT *re-translate* canonical identifiers that are latin in both languages (§1.3) — `BR` stays `BR` , never an English expansion used as the identifier.
4. **Closed-list canonical terms** — §1.9 fixes the closed list of canonical RENAR terms (mirrors `standard/04`). Coining new project-local terms is forbidden.
5. **Single source of truth** — the canonical normative source is `standard/04-terms.md` . `reference/01-glossary.md` (and its EN counterpart) is an informative companion.
6. **Domain-context preservation** — substrate / VCS / SE technical terms (`commit` , `merge` , `hook` , `pipeline` , `runner` , `linter`) stay latin: they are industry conventions in English too.

§1.2 Five buckets

Every term in the EN normative corpus is classified into one of five buckets:

Bucket	Category	Policy	Section
A	Canonical identifiers (IDs, types, statuses, field names)	Keep latin verbatim — shared with RU	§1.3
B	RENAR conceptual terms (RU Style Guide §1.15 reversal)	Render in native English	§1.4
C	Technical loanwords / SE vocabulary	Keep native English form	§1.5
D	Role names	Canonical English role names	§1.6
E	External-standard terms	Keep the source standard's spelling	§1.7

> *The RU Style Guide has six buckets because its central concern is which Russian-vs-latin form to use. The EN corpus is natively English, so its buckets instead govern what must stay latin (Bucket A) and what must read as native English (Buckets B–E). The reverse-calque guard (§1.1.3) replaces the RU corpus's anglicism guard.*

§1.3 Bucket A — canonical identifiers (keep latin verbatim)

Rule: these stay latin, identical to the RU corpus, in prose, code-fence content, frontmatter, and section labels. They are machine-readable identifiers; translating them breaks the referential chain across both language editions.

Class	Members
Artifact / requirement types	`BR`, `SR`, `SPEC-*` (`SPEC-UI`, `SPEC-AI`, `SPEC-INT`, `SPEC-ARCH`, `SPEC-OPS`, `SPEC-DATA`, `SPEC-API`, `SPEC-PROC`, `SPEC-SEC`), `TC`
ADAPT family	`ADAPT`, `delta-TZ`, `trigger-stage`
TZ	`TZ` and the ID format `TZ-YYYY-NNN` (see §1.10)
Quality Gates	`QG-0`, `QG-1`, `QG-2`, `QG-3`, `QG-4`
Maturity / capabilities	`RENAR-1` .. `RENAR-5`, `V1` – `V6`, `MVR-*`
Lifecycle statuses	`draft`, `proposed`, `approved`, `verified`, `accepted`, `done`, `obsolete`, `deprecated`, `frozen`, `active`, `planning`, `blocked`, `review`, `ready`, `passing`, `failing`, `superseded`, `answered`, `resolved`, `revised` (§1.3.1)
Field names / YAML keys	`parent:`, `verifies[]`, `source.tz-section`, `source.adapt`, `adversarial-review-ref`, `ai-provenance`, `audit-trail`, `declared-stricter`, `assessment-mode`, `supersedes`, `superseded-by`, `role:`, `status:`, `lang:`
Proper nouns	`RENAR`, `SENAR`, `ISO`, `IEC`, `IEEE`, `RFC`, `GDPR`, `SAFe`, `BABOK`, `GitHub`, `Astro`, `Markdown`
Substrate / VCS / SE	`commit`, `merge`, `diff`, `branch`, `push`, `pull`, `hook`, `patch`, `frontmatter`, `slug`, `hash`, `runner`, `pipeline`, `release`, `deploy`, `build`, `workflow`, `linter`, `parser`, `compiler`, `tracker`

§1.3.1 Status case consistency

Lifecycle statuses are always lowercase, even at sentence start where possible (reword to avoid leading a sentence with a status). `Approved` / `APPROVED` as a status token is a paste-error; the value is `approved`.

§1.4 Bucket B — RENAR conceptual terms (native English)

These terms are rendered in **Russian prose** in the RU corpus (RU Style Guide §1.15). In the EN corpus they revert to their **native English** form. The latin identifiers in the right column stay latin in both languages.

EN canonical (prose)	RU prose (reference only)	Stays latin in
manifest; conformance manifest	манифест; манифест соответствия	file name `RENAR-CONFORMANCE.yaml`, field names
conformance; conformant; non-conformant	соответствие; соответствующий; несоответствующий	`RENAR-CONFORMANCE.yaml`, `senar-version` / `renar-version`, levels `RENAR-1..5`
provenance	происхождение	field `ai-provenance`, `*-provenance`
adversarial review; adversarial reviewer;	состязательный обзор; состязательный рецензент; состязательный критик	field `adversarial-review-ref`

adversarial critic		
lifecycle; lifecycle state / status	жизненный цикл; состояние/статус жизненного цикла	drift class `Lifecycle drift`, status values, file names
mandatory clauses / mandatory clause; mandatory (adj.)	обязательные положения / обязательное положение; обязательный	schema annotation `mandatory` in YAML
enforcement; enforcement point	обеспечение соблюдения; точка контроля	field / identifier names
Source-of-Truth inversion (SoT inversion); Source of Truth (SoT)	инверсия источника истины; источник истины	—
canonical (adj.)	канонический	bilingual headers `English (canonical)`
closed list	закрытый список	—
state machine	машина состояний	—
data model; multilingual; spot-check; normative; trace chain; full-completeness	модель данных; многоязычный; выборочная проверка; нормативный; цепочка прослеживаемости; полнота	drift classes (`Order / provenance drift`), external terms

Rule: prefer the established English term over a literal calque of the Russian prose. `Source of Truth` (not "source of verity"), `provenance` (not "origin-tracking"), `state machine` (not "states automaton").

§1.5 Bucket C — technical loanwords / SE vocabulary (native English)

Standard requirements-engineering and software-engineering vocabulary is used in its ordinary English form — no special treatment, no quoting:

`scope`, `audit`, `policy`, `ownership`, `default`, `bypass`, `rollback`, `override`, `stub`, `walkthrough`, `quickstart`, `baseline`, `traceability`, `immutable`, `approval`, `findings`, `eval / evaluation`, `version pin`, `branching`.

Rule: these read as plain English. The RU corpus debates whether to keep them latin or rewrite them; in EN there is no debate. Hyphenated identifier forms (`audit-trail`, `baseline-dataset`) stay latin per Bucket A.

§1.6 Bucket D — role names (canonical English)

SENAR §4 role names are canonical in English; the RU corpus translates them (RU Style Guide §1.15). The EN canonical forms:

EN canonical	RU prose (reference only)	Notes
Architect	Архитектор	—

Engineer	Инженер	—
Reviewer	Рецензент	the role; lowercase `review` is the lifecycle status
Supervisor	Супервизор	—
Stakeholder	Заинтересованная сторона	`Stakeholder Requirement` is the BABOK term
Product Owner	Владелец продукта	Scrum/SAFe term

The `role:` field value (`authorized-role-holder` , etc.) stays latin in both languages.

§1.7 Bucket E — external-standard terms

Terms owned by an external standard keep that standard's spelling and casing: `Stakeholder Requirement` (BABOK), `Statement of Need` (ISO 29148), `Definition of Done` (Scrum), `Product Backlog` (Scrum/SAFe). Do not paraphrase them into RENAR vocabulary.

§1.8 Reverse-calque failure modes

A translator working from RU is prone to:

- **Literal syntax calque** — preserving RU word order or predicate chains (`X is being represented by Y` for `X — это Y`). Use idiomatic English: `X is Y` .
- **Transliteration leakage** — leaving a transliterated Russian term (`adapt-irovanie`) instead of the English (`adaptation`).
- **Over-translation of identifiers** — expanding `BR` to "Business Requirement" *as the identifier*. The identifier is `BR` ; "Business Requirement" is its gloss, used once at definition.
- **Modal flattening** — collapsing RU lowercase modals to indicative English and losing the RFC-2119 level (see §2). The EN form MUST be an UPPERCASE keyword.

Mitigation: the §1.4 reversal table is the hard-coded substitution set; semantic preservation (§2.5) is an explicit check; a human spot-check on a random 10% of edits is required.

§1.9 Closed-list canonical RENAR terms

The canonical list lives in `standard/04-terms.md §4.3–§4.7` (artifacts, the SPEC family, TC types, Quality Gates, lifecycle statuses, V1–V6 capabilities, drift classes). The Style Guide reflects this canonical list; it does not duplicate it.

§1.10 TZ — the source requirements artifact

Decision #80. Russian `ТЗ` is rendered in EN as `TZ` (latin), not translated to "ToR" / "SoW" / "Technical Assignment". Rationale: `TZ` is already a latin canonical identifier in both language editions — the artifact ID format is `TZ-YYYY-NNN` (`standard/04 §4`), the field is `source.tz-section` , the directory is `tz/` , and the pointer format is `[TZ-XXX §Y]` . Translating the prose mention while keeping the latin identifier would split a single concept across two spellings.

Gloss (used once, at first mention in the EN glossary): *TZ — the immutable contractual requirements artifact (ID `TZ-YYYY-NNN`); the client's source-of-record requirements document, adapted into RENAR*

artifacts via ADAPT.

§1.11 Adding new terms

1. **Identify need** — a term appears in an EN normative chapter ≥3 times without a §1.9 listing.
2. **Bucket classification** — assign A–E per §1.2.
3. **Rationale draft** — 3–5 sentences: why needed, why this bucket, what alternatives were rejected.
4. **Review** — Style Guide editor (or project owner), same procedure as for a `standard/04` formal change.
5. **Retroactive** — after approval, existing occurrences are normalized.

---

§2 RFC-2119 EN normative wording

§2.1 Principles

1. **Semantic-fidelity first** — RFC-2119 (RFC 2119) fixes strict discrimination between `MUST` / `SHOULD` / `MAY`. EN wording MUST preserve this discrimination.
2. **No semantic downgrade** — an editorial pass MUST NOT change a modal-verb level. `MUST` → `SHOULD` is a **content change** that requires a separate task.
3. **EN UPPERCASE canonical** — RFC-2119 keywords in EN normative clauses are written in **UPPERCASE** (`MUST` , `MUST NOT` , `SHOULD` , `SHOULD NOT` , `MAY` , `REQUIRED` , `RECOMMENDED` , `NOT RECOMMENDED` , `OPTIONAL` , `SHALL` , `SHALL NOT`). This is the canonical normative weight per RFC 2119 §1 and RFC 8174.
4. **Polarity inversion vs RU** — the RU corpus uses **lowercase** modals as canonical (RU Style Guide §2.2, Option C, under the RFC 8174 carve-out). That carve-out is RU-only and **does not** extend to EN. The EN corpus uses UPPERCASE; lowercase English modals (`must` , `should`) in prose carry **no** normative weight and are reserved for descriptive sentences.
5. **Imperative mood for normative clauses** — normative clauses use the active/imperative form (`An SR MUST trace to its parent BR`). Indicative descriptive prose uses lowercase verbs.
6. **RU lowercase only in citation** — RU modal verbs (`должен` , `следует` , `может` , …) appear in the EN corpus **only** inside an explicit citation or a bilingual mapping (e.g. quoting the RU canonical wording). They are never used as the EN corpus's own normative wording.

§2.2 Regime decision (EN UPPERCASE canonical)

The RENAR Standard EN corpus uses **UPPERCASE RFC-2119 keywords** as canonical normative wording. This is the international default for normative specifications (IETF, W3C, ISO-adjacent practice) and removes ambiguity per RFC 8174.

§2.3 Canonical mapping table

11 RFC-2119 / RFC-8174 keywords ↔ EN canonical (UPPERCASE) ↔ RU canonical (lowercase, citation only). **Bidirectional usage:** EN translation — RU lowercase → EN UPPERCASE; RU pass — EN UPPERCASE → RU lowercase.

Mandatory levels

RFC-2119 (canonical EN)	EN synonyms	RU equivalent (citation only)	Semantics
`MUST`	`REQUIRED`, `SHALL`	должен / обязан	Absolute requirement
`MUST NOT`	`SHALL NOT`	не должен / запрещено	Absolute prohibition
`REQUIRED`	`MUST`	обязателен / требуется	Equivalent to MUST (adjective form)

Strong-recommendation level

RFC-2119 (canonical EN)	EN synonyms	RU equivalent (citation only)	Semantics
`SHOULD`	`RECOMMENDED`	следует / рекомендуется	Strong recommendation; exceptions require explicit rationale
`SHOULD NOT`	`NOT RECOMMENDED`	не следует / не рекомендуется	Strong negative recommendation

Optional level

RFC-2119 (canonical EN)	EN synonyms	RU equivalent (citation only)	Semantics
`MAY`	`OPTIONAL`	может / допускается	Truly optional; no preference

Negation discrimination

Form	EN canonical	Discrimination
MUST NOT / SHALL NOT	`MUST NOT`	Absolute prohibition
SHOULD NOT / NOT RECOMMENDED	`SHOULD NOT`	Strong negative recommendation
(no canonical "MAY NOT")	"need not" / "is OPTIONAL"	RFC 2119 has no MAY NOT; "need not" = "MAY omit"

Hard rule: `MUST NOT` ≠ `SHOULD NOT`. The editor never substitutes one for the other "for readability".

§2.4 Tense / mood normative rules

§2.4.1 Normative clauses → imperative / active

```
# Canonical
An SR MUST trace to its parent BR through the `parent:` link.
```

```
# Anti-pattern (indicative — normative weight lost)
An SR traces to its parent BR through the `parent:` link.
```

§2.4.2 Definitions → predicative form

Canonical form: `X is Y, governed by Z` or, in a glossary, `X — Y`.

```
ADAPT is the two-way adaptation of a TZ, governed by the client-agreement
process.
```

Anti-pattern: `X represents/constitutes Y` chains (calque overhead); `X is being defined as Y`.

§2.4.3 Future tense — scope-limited

Future tense (`will`) is **not** used in normative clauses. It is permitted only in roadmap sections, forward-looking limitations, and conditional consequences.

```
# Wrong (ambiguous — requirement or forecast?)
An SR will contain a parent link.

# Correct
An SR MUST contain a parent link.
```

§2.4.4 Conditional clauses

Canonical pattern: `If X, then Y MUST Z`. Variant: `When X, Y MUST Z`.

Anti-pattern: conditional + indicative (`If X, Y does Z`) — normative weight lost.

§2.5 Strict semantic-preservation rule

Hard rule: an editorial pass MUST NOT change an RFC-2119 modal-verb level.

Permitted (semantic-preserving):

From	To	Reason
`MUST` ↔ `REQUIRED`	—	Same level (verb vs adjective form)
`MUST` ↔ `SHALL`	—	Equivalent per RFC 2119 §1
`SHOULD` ↔ `RECOMMENDED`	—	Same level
`MAY` ↔ `OPTIONAL`	—	Same level
`MUST NOT` ↔ `SHALL NOT`	—	Same level

Forbidden (content change, not editorial):

From	To	Reason
`MUST`	`SHOULD`	MUST → SHOULD downgrade
`SHOULD`	`MAY`	SHOULD → MAY downgrade
`MUST NOT`	`SHOULD NOT`	MUST NOT → SHOULD NOT downgrade
Any modal	indicative (`X does Y`)	Loss of normative weight

Verification protocol: pre-pass, count modal keywords per level in the chapter; post-pass, re-scan; the per-level sum MUST NOT change. If it changes, flag as semantic drift, revert, and raise a content-review task. The RU↔EN modal counts are reported by `scripts/check-rfc-modals.js` (dual RU-lowercase / EN-UPPERCASE inventory).

§2.6 Descriptive carve-out

When a passage *describes* RFC-2119 vocabulary rather than *using* it normatively (e.g. "RFC 2119 defines `MUST` as an absolute requirement"), the keyword is a descriptive mention. Such mentions are permitted in any section, provided the keyword is wrapped in a code-fence, backticks, or a blockquote / citation. A descriptive mention does not carry normative weight.

---

§3 Formatting conventions

These conventions are largely shared with the RU Style Guide (§3); EN-specific deltas are typography (§3.7). Structural rules (headings, code-fences, tables) are language-agnostic.

§3.1 Principles

1. **Substrate readability first** — preserve git-blame readability; do not break syntax highlighting; stay stable under native substrate processing (markdown lint, frontmatter validator).
2. **Scan-readable second** — numbered sections for cross-ref, consistent bullets, typographic punctuation.
3. **No tool dependency** — conventions are applicable in any text editor.
4. **Accessibility** — semantic heading hierarchy (no H1 → H3 skip); typed code-fences; alt-text (deferred).
5. **Structural parity with RU** — the EN file mirrors the RU original's section numbering and anchor structure so cross-references and parity checks line up.

§3.2 Citations & cross-refs

Three classes:

Class	Pattern	When
(A) Bare intra-file	`§3.5`	Reference within the same file
(B) Markdown link cross-file	`[§4.5](../../standard/en/04-terms.md#4.5)`	Reference to another file

(C) Anchor-only intra-file	`[§3.5](#35-tables--lists)`	Clickable intra-file reference

Hard rule: a cross-file reference MUST be a markdown link (Class B). A bare `§X.Y` for a cross-file target is an anti-pattern (not clickable, no PDF outline integration).

Cross-edition links. An EN file SHOULD link to the EN counterpart of its target where that counterpart exists (`../../standard/en/04-terms.md`). Until the EN counterpart exists, link to the canonical RU source (`../../standard/04-terms.md`), which is the source of truth; the `en-crosslinks-changelog` pass re-points these once the EN corpus is complete.

"See" (EN) is used; "См." (RU) is not used in EN files.

Anchor format: lowercase, hyphens instead of spaces / dots. Verified post-edit by `scripts/check-md-links.js` .

§3.3 Code-fences

Hard rule: every fence MUST have a language tag. No-lang fences are an anti-pattern.

Language-tag whitelist (closed list): `yaml` , `bash` , `cypher` , `markdown` , `json` , `python` , `sql` , `text` .

Inline code (backticks): identifiers (`` `SR-001` ``), file paths (`` `standard/04-terms.md` ``), type / field names (`` `parent: BR-001` ``), short CLI fragments.

Anti-pattern: backticks around whole sentences — reserve backticks for technical identifiers.

§3.4 Headings

§3.4.1 Numbering convention

Directory	Convention
`standard/en/`	Dotted-number (`## 3.5 Tables / lists`) — mirrors RU
`reference/en/`	Dotted-number
`core/en/`	Plain (pedagogical narrative)
`guide/en/`	Plain or numbered phase walkthroughs

§3.4.2 Depth rule

- Max normative depth — **H3**.
- H4 — exception (substructure inside a large H3), ≤ 5 per chapter.
- H5 / H6 — **banned** (restructure the parent instead).
- Depth jumps (H1 → H3 without an intermediate H2) — **banned**.

§3.4.3 Casing & form

Headings are **sentence-case** (capitalize the first word and proper nouns only), not Title Case. A heading is a short label, not a sentence: `## Tables and lists` , not `## This Section Describes Tables` .

§3.5 Tables / lists

Hard rules:

- Bullet style — `-` only. `*` / `+` — banned. Mixed bullets within a file — banned.
- Table alignment — `---` only. `:---` / `---:` / `:---:` — banned (renderer-dependent variance).
- Nested unordered — 2-space indent (CommonMark default).
- Nested ordered — 3-space indent.
- Ordered list — manual `1.` , `2.` , `3.` preferred (explicit count visible in source).

§3.6 frontmatter YAML

Canonical field order:

```yaml
---
title: "Document title"
description: "Optional one-line summary"
order: 6
lang: en
version: "1.0"
status: draft  # | proposed | approved | verified | obsolete | frozen
---
```

Quoting policy: strings with spaces / special chars use double quotes; pure ASCII identifiers (`status: draft` , `lang: en`), ISO dates, numerics, and booleans are unquoted.

`lang:` **closed list** — `ru` , `en` . Every EN file MUST declare `lang: en` . This is the single signal RU-targeted gates use to scope themselves out (§4.2).

Validation: `scripts/validate-frontmatter.js` .

§3.7 EN typography

§3.7.1 Quotes — straight double quotes

Canonical: straight ASCII double quotes `"..."` for outer quotes in EN prose. (The RU corpus uses guillemets `«»` ; the EN corpus does not.) Curly quotes are not required; ASCII is the corpus default and is git-blame-stable. The RU-only ASCII-quote check does not apply to EN files.

§3.7.2 Em-dash — `—`

Em-dash `—` (U+2014) for parenthetical separators and definition predicates (`X — Y`). Hyphen `-` for compound words (`closed-list` , `source-of-truth`), file paths, and identifiers. En-dash `–` (U+2013) for number ranges (`§3.5–3.7`).

Anti-pattern: `-` or `--` in place of `—` .

§3.7.3 Numbers

Form	Convention
Whole numbers ≤ 999	No separator: `42`, `999`
Whole numbers ≥ 1000	Comma separator (EN convention): `1,000`, `50,649`
Percentages	`33%` (no space)
Decimals	Period: `0.5`
Ordinals	`1st`, `2nd` (EN suffix)

> ***Delta from RU:*** *the RU corpus uses a space thousands-separator (`1 000`) and RU ordinal suffixes (`1-й`). The EN corpus uses the comma separator (`1,000`) and EN ordinals (`1st`).*

§3.7.4 Math operators

`≥` / `≤` / `≠` / `±` / `→` / `↔` — Unicode preferred in prose. ASCII `>=` / `<=` / `!=` / `->` are anti-patterns in prose; acceptable inside code-fences.

§3.8 Cross-file link conventions

- **Relative paths only** (from the linking file's directory). From `reference/en/06-en-style-guide.md`, the repository root is `../../`.
- **Anchor case:** lowercase, hyphens.
- **Dead-link prevention:** the site build and `scripts/check-md-links.js` flag broken links.

```
# from reference/en/06-en-style-guide.md
[§4.5 standard/04](../../standard/en/04-terms.md#4.5)
[reference sibling (RU)](../01-glossary.md)
```

§3.9 Inline emphasis (bold / italic)

Bold (`**text**`) — for term introduction (`**Closed list policy** — the formal mechanism …`), emphatic normative flags (`**Hard rule:** …`), and anti-pattern flags (`**Anti-pattern:** …`).

Hard rule: bold is **not** a replacement for UPPERCASE RFC-2119 keywords. In EN, the keyword itself is UPPERCASE; bold is additive emphasis, not normative weight.

Italic (`*text*`) — restricted: foreign-language phrases (`*ad hoc*`, `*de facto*`), titles of external works (`*Software Requirements* (Wiegers)`). Not for general emphasis (use bold).

---

§4 Integration & enforcement

§4.1 Canonical sources

Source	Purpose
`standard/04-terms.md`	Canonical RENAR terminology (closed list from §1.9)
`standard/01-scope.md §1.7`	Master index of closed lists
`reference/01-glossary.md`	Informative glossary (EN counterpart: `reference/en/01-glossary.md`)
`reference/02-schemas.md`	Canonical frontmatter schemas
`reference/06-ru-style-guide.md`	RU Style Guide — the mirror source of this guide

§4.2 Automated enforcement (scripts/)

The Style Guide sets the rules; these scripts enforce them. EN files are scoped via the `lang: en` frontmatter signal.

Script	Behaviour on EN files	Style Guide §
`scripts/validate-frontmatter.js`	Validates `lang: en`, required fields, closed value lists	§3.6
`scripts/check-rfc-modals.js`	EN UPPERCASE modal inventory (RU lowercase for RU files)	§2.1, §2.5
`scripts/check-substrate-term.js`	**Skipped** (EN uses native `substrate`)	§1.5
`scripts/check-process-vocab.js`	**Skipped** (EN uses native English process vocab)	§1.5
`scripts/style-guide-check.js`	RU-prose checks (bucket-a, bucket-e, en-uppercase, ascii-quote, anglicism-label) **skipped**; `fence-lang` active	§1, §3.3
`scripts/check-md-links.js`	Active (language-agnostic)	§3.2, §3.8
`scripts/check-section-refs.js`	Active (language-agnostic)	§3.2
`scripts/check-en-parity.js`	EN↔RU structural parity (heading counts, counterpart presence)	§0.3, §3.1

npm scripts: `npm run check:all` runs the full gate sweep (RU + EN scoped).

§4.3 Change procedure (cross-ref)

See §0.5 Change procedure.

§5 Limitations & follow-ups

- **EN glossary** — `reference/en/01-glossary.md` (task `en-glossary`) is the term-level source of truth for translating agents; this guide governs form, the glossary governs term equivalents.
- **Cross-edition links** — until the EN corpus is complete, EN files link to canonical RU sources; the `en-crosslinks-changelog` pass re-points them (§3.2).
- **Imagery / SVG alt-text** — deferred until images enter the corpus.
- **Curly quotes** — ASCII straight quotes are the corpus default; a future amendment may adopt curly quotes if a renderer benefit is established.

§6 Sources

- **RENAR Standard** — `standard/`.
- **RU Style Guide** — `reference/06-ru-style-guide.md` (mirror source).
- **SENAR (parent standard)** — methodological base.
- **RFC 2119** — Key words for use in RFCs (1997).
- **RFC 8174** — Ambiguity of Uppercase vs Lowercase (2017).
- **ISO/IEC/IEEE 29148:2018** — Requirements engineering.

RENAR EN Style Guide v1.0 — EN translation epic foundation, 2026-06-06. Mirror of the RU Style Guide (`reference/06`) with inverted RFC-2119 polarity (EN UPPERCASE canonical).

ISO/IEC 29148 — Trace Matrix

> ***Purpose:*** *verifiable conformance of a conformance claim to ISO/IEC/IEEE 29148:2018 (standard/14 §14.4.2). Normative field definitions are in standard/06, standard/08, standard/09, 02-schemas.md.*

RENAR simplifies the set of mandatory 29148 attributes (18 → 7–8 per artifact) and adds TC as a first-class artifact, ADAPT, and the SPEC axis. The table below is the **complete** trace for a conformance assessor and for populating `external-claims[]` in the manifest.

1. Requirement classes (29148 §5)

ISO/IEC 29148 class	RENAR artifact	Normative source
Stakeholder requirement	`BR`	§6.5
System requirement	`SR` (level: system / subsystem / module)	§6.6
Software requirement (implementation unit)	`TR`	§6.7
Interface / design specification	`SPEC-*` (11 types)	§8
Verification item	`TC`	§9
Requirements validation (agreement with the client)	`ACTZ` + `AT`	§7.13, §9.19
TZ interpretation (internal artifact; an extension of 29148)	`ADAPT`	§7

2. Requirement attributes (29148 Table B.1 → RENAR)

#	ISO/IEC 29148 attribute	RENAR field / mechanism	Mandatoriness	Note
1	Unique ID	`id` (immutable)	mandatory	V1; see §3.3.1
2	Requirement statement	body (Need / Behavior / ...)	mandatory	EARS template for SR: §6.6.3
3	Rationale	body "Context" + `source.adapt-section` (when an ADAPT exists) or `source.adversarial-review-ref`	mandatory (BR/SR)	Traceability to ADAPT, or to the AR under a "no findings" verdict (§7.4.1)

4	Source	`source.tz-section` (always); `source.adapt` **or** `source.adversarial-review-ref` (conditional, §7.4.1); `source.document-ref`	mandatory	V5 pinning via `document-ref`
5	Fit criterion	body "Success criteria" (BR) / Pass criteria of TC	mandatory	Measurability via 25022/25023: §14.4.4
6	Priority	`priority` (MoSCoW)	mandatory	WSJF — informative in the SAFe mapping
7	Owner	`owner` (BR/SR) / `business-context.stakeholder`	mandatory	§6.5.2
8	Status	`status` (lifecycle enum)	mandatory	State machines: §10
9	Verification method	`verified-by[]` → `TC` + `tc-type`	mandatory for verify	TC as a first-class artifact — an extension of 29148
10	Parent / child	`parent.id`, auto `children[]`	mandatory (SR/TR)	Hierarchy BR→SR→TR
11	Traceability (derived)	`verified-by`, `constrained-by[]`, `implements-spec[]`, KG edges	derived	reference/05 §3
12	Version	substrate-native version + `requirement-version` in TC	mandatory (V5)	§3.3.5
13	Author	V6 author + `ai-provenance`	mandatory (RENAR-4+ AI)	§4.10.1
14	Date created / modified	substrate change-record timestamps	derived (V6)	Audit log: §10.13
15	Risk	`compliance[]`, AIR register link	optional / domain	03-ai-risk-register.md
16	Assumption	ADAPT backward findings `type: hidden-assumption`	via ADAPT	§7.4.4
17	Dependency	`depends-on[]` (SPEC), `constrained-by[]` (SR)	mandatory where applicable	DAG invariant: 02 §9
18	Approval authority	QG-0 / QG-2 + ADAPT architect signature + ACTZ dual signature	mandatory	Replacement for the formal walkthrough: §14.4.2

Not adopted from 29148: review meetings and inspection-only verification without TC evidence — see §14.7.

3. Verification methods (29148 §6.4)

29148 method	RENAR implementation
Test	`TC` with `tc-type: system | business | contract | …`
Demonstration	`AT` against the effective TZ (§9.19); `tc-type: business` + QG-4 (optional)
Inspection	`[test-spec-change]` workflow + adversarial review (§9.13)
Analysis	SR with `quality-characteristic` + eval-TC (`tc-type: eval`) for SPEC-AI

4. Lifecycle processes (29148 §6)

29148 process	RENAR chapter	Gate
Requirements elicitation	ADAPT backward + TZ	QG-3 (ADAPT approve, §10.4.1)
Requirements analysis	BR/SR decomposition	QG-0 (BR/SR approve)
Requirements specification	SPEC axis	QG-3 Architecture (optional/required)
Requirements verification	TC + QG-2	QG-2 Verification
Requirements validation	A signed ACTZ (§7.13) + AT against the effective TZ	AT release gate (§10.4.3) — mandatory and not a Quality Gate; QG-4 (optional) measures the business outcome
Requirements management	lifecycle §10 + substrate V1–V6	Continuous

5. Use in conformance assessment

1. For each `ISO/IEC/IEEE 29148:2018` claim in the manifest — walk through the rows of §2–§5.
2. By spot-check (≥10% of artifacts, or all system-level BR/SR) verify the presence of mandatory fields and the `source.adapt` trace.
3. Non-conformance of any **mandatory** row of §2 — a partial claim is not permitted (§14.6.2).

Reference RENAR 1.0 — renar.tech

Conformance Self-Assessment

> ***Purpose:*** *a practical kit for the Tech Lead / Architect before releasing RENAR-CONFORMANCE.yaml. The normative basis is standard/13. This document is* ***informative****; on conflict, standard/13 wins.*

1. MVR ↔ mandatory clauses §13.3 bijection

MVR (§0.5)	§13.3 clause	`mandatory-clauses-confirmed` field
MVR-1 Source-of-Truth inversion	§13.3.1	`sot-inversion: true`
MVR-2 V1–V6	§13.3.2	`substrate-v1-v6: { v1..v6: true }`
MVR-3 ADAPT per TZ	§13.3.3	`adapt-per-tz: true`
MVR-4 11 SPEC types	§13.3.4	`spec-types-closed-list: true`
MVR-5 TC pos/neg	§13.3.5	`tc-pos-neg-pairing: true`
MVR-6 QG — closed list	§13.3.6	`quality-gates-closed-list: true`
MVR-7 conformance manifest	§13.4 (artifact)	manifest exists + signed
— (closed-list policy)	§13.3.7	`closed-lists-backward-findings: true`

All seven MVR + §13.3.7 are mandatory for **any** RENAR-1..5 level.

2. Self-assessment checklist (mandatory clauses)

Check off after verifying the evidence in the substrate:

§13.3.1 Source-of-Truth inversion

- ☐ The BR/SR/SPEC/TC hierarchy is the authoritative source of behavior
- ☐ No SR reconstructed from code without a defect-fix justification
- ☐ Drift-hooks / review policy block silent SR←code adaptation

§13.3.2 V1–V6 capabilities (substrate)

- ☐ V1 immutable history — enabled
- ☐ V2 atomic change unit — enabled
- ☐ V3 diff & review — enabled
- ☐ V4 branching / change-set — enabled
- ☐ V5 end-to-end version pinning across substrates — enabled (`verifies[].requirement-version`)
- ☐ V6 author + timestamp — enabled

§13.3.3 Reactive ADAPT

- [ ] Every TZ has passed the adversarial review; the outcome is issued as an AR in status `issued`
- [ ] On a "findings present" verdict: the ADAPT is in status `approved` with the Architect's signature
- [ ] On a "no findings" verdict: no ADAPT is created; BR/SR/SPEC carry `source.tz-section` + `source.adversarial-review-ref`
- [ ] Every finding in `resolved` carries `decided-in` on a clause of a signed ACTZ
- [ ] Signed ACTZ carry the dual signature (client + contractor)
- [ ] A client signature on an ADAPT is not declared mandatory (admissible only as `declared-stricter`)
- [ ] delta-TZ: the delta-ADAPT is created on the fact of findings from the delta-TZ review, not automatically
- [ ] Superseded ADAPT are retained in the terminal `superseded`; no dangling `source.adapt` remain

§13.3.4 SPEC types

- [ ] All SPEC ∈ {ARCH, API, DATA, INT, PROC, UI, AI, SEC, OPS, TEST, DOC} — the closed list of 11 types
- [ ] No local `SPEC-CUSTOM-*`
- [ ] Test benches and data are described as `SPEC-TEST`, not as a section of `SPEC-OPS`
- [ ] Delivered documentation is described as `SPEC-DOC`; the internal runbook stays in `SPEC-OPS`
- [ ] Every `SPEC-TEST` in which at least one dataset carries real or personal client data has a `client-signature`

§13.3.5 TC pos/neg

- [ ] Every verifiable assertion has a pos + neg TC (or a negative-invariant exception)
- [ ] QG-2 blocks `verified` on violation
- [ ] Every TC with `automation.kind: dynamic` carries an `environment-ref` to an existing `SPEC-TEST` (a missing field is fatal; static checks require no bench)
- [ ] Every `SPEC-DOC` is covered by the doc-lint (a TC with `automation.kind: static`); without it the SPEC-DOC does not pass QG-2

§13.3.6 Quality Gates

- [ ] QG-0, QG-1, QG-2 implemented as `required`
- [ ] QG-3, QG-4 declared `required` | `declared` | `absent` in the manifest
- [ ] No local custom gates

§13.3.7 Closed lists

- [ ] Backward finding types — closed list §7.4.4 only
- [ ] SPEC decomposition types — closed list §8 only

Rule: if at least one is unchecked → **do not release** the manifest (§13.5.1).

3. Level checklist (choose the target RENAR-N)

The minimum for claiming a level is standard/11 §§11.4–11.8. Brief summary:

Level	Key additional criteria
RENAR-1	Mandatory clauses only; frontmatter minimal
RENAR-2	Basic frontmatter (no strict schema validation); TZ and delta-TZ as explicit immutable artifacts; lifecycle statuses not yet mandatory
RENAR-3	Full frontmatter schema + lifecycle statuses; hooks enforce QG-0 and QG-1, QG-2 — partially
RENAR-4	ai-provenance mandatory; pos/neg pairing for every normative assertion; QG-2 enforced natively
RENAR-5	Adversarial review as a gate; multi-model consensus for `priority: must`; knowledge graph as primary lookup; continuous evaluation of SPEC-AI

- [ ] The chosen `level` in the manifest is **no higher** than the checklist actually passed
- [ ] `declared-stricter` (if present) is documented separately

4. Manifest template (minimal)

Save as `RENAR-CONFORMANCE.yaml` at the root of the requirements substrate:

```yaml
manifest-version: 1
manifest-id: "CFM-YYYY-NNN"
renar-version: "1.0"
senar-version: "1.0"
level: RENAR-2
assessment-date: "2026-05-22"
assessment-mode: self          # self | third-party (§13.4.2)
next-assessment-due: "2026-08-22"

mandatory-clauses-confirmed:
  sot-inversion: true
  substrate-v1-v6: { v1: true, v2: true, v3: true, v4: true, v5: true, v6: true }
  adapt-per-tz: true
  spec-types-closed-list: true
  tc-pos-neg-pairing: true
  quality-gates-closed-list: true
  closed-lists-backward-findings: true
```

```yaml
quality-gates:
  qg-0: required
  qg-1: required
  qg-2: required
  qg-3: declared
  qg-4: absent

external-claims:
  - standard: "ISO/IEC/IEEE 29148:2018"
    clause-ref: "§14.4.2"
    claim: "partial"           # full | partial | aligned

substrate-capabilities:
  v1-immutable-history: declared
  v2-atomic-change-unit: declared
  v3-diff-review: declared
  v4-branching: declared
  v5-version-pin: declared
  v6-author-timestamp: declared
  substrate-id: "<substrate-native pointer>"

spec-types-supported: ["SPEC-ARCH", "SPEC-API", "SPEC-DATA", "SPEC-INT",
                       "SPEC-PROC", "SPEC-UI", "SPEC-AI", "SPEC-SEC", "SPEC-OPS",
                       "SPEC-TEST", "SPEC-DOC"]

assessor:
  id: "<V6 author identifier>"
  role: architect              # architect | authorized-role-holder | external-
assessor
  signature-ref: "<substrate-native pointer to the signature event>"
```

The full field list is §13.4.2.

5. Cadence

- Self-assessment: **quarterly** (default)
- After a delta-TZ that affects mandatory clauses — **out of cycle**
- Conformance-loss triggers — §13.8

Reference RENAR 1.0 — renar.tech

Pedagogical Density (WC-02)

> ***Informative.*** *A density rating of the normative text for calibrating the reader's route. It does not change any mandatory clauses.*

Method (2026-05-22): `density score = (RFC-2119 RU markers + closed-list refs + state machine tables) / 1000 lines`, manual calibration + line counting. Scale: **L** (low ≤3), **M** (3–6), **H** (6–9), **VH** (≥9).

By Chapter (density ratings)

Chapter	Lines	Score	Tier	Signpost
00 Introduction	122	4.2	M	standard/README → start with core
01 Scope	221	5.8	M	closed lists §1.7 — see glossary
02 Methodology	218	5.0	M	Source of Truth — core "How RENAR Works" first
03 substrate V1–V6	245	5.2	M	→ guide/03 or guide/04
04 Terms	425	7.2	H	→ reference/01 for examples
05 Roles	242	4.5	M	→ guide/01 walkthrough
06 Hierarchy	678	8.4	H	→ guide/00 quickstart before frontmatter
07 ADAPT	632	6.8	H	→ core ADAPT section
08 Specifications	489	7.9	H	→ guide/09 E3 SPEC examples
09 Test cases	732	9.1	VH	→ reference/02 §8 + §9.1.1 decision tree
10 Lifecycle QG	703	9.5	VH	→ §10.1.1 QG tree + guide/00
11 Maturity	304	4.8	M	→ guide/02 transition
12 Metrics	209	3.9	M	provisional targets §12.4
13 Conformance	399	8.0	H	→ reference/08 kit first
14 Normative refs	~220	2.8	M	→ reference/11 extended catalogue; §14.5 "the five key ones"

Top 5 "Hotspots" (signposted)

1. **§10 Lifecycle** — see quickstart QG flow
2. **§9 TC** — see schemas §8 + adversarial §guide/07
3. **§6 Hierarchy** — see quickstart + walkthrough
4. **§13 Conformance** — see reference/08 kit

5. **§8 SPEC** — see guide/09 E3

Agent Implementation Profile

> ***Purpose:*** *an informative contract for an agent or substrate-native runtime that* ***implements*** *RENAR without guesswork. The normative text is* `standard/` *; this document is an operational mapping with no ties to any specific vendor tooling.*
> ***Machine-readable:*** `normative-index.yaml`

1. How to Read the Table

Column	Meaning
`clause_id`	Stable ID from `normative-index.yaml`
`Agent action (abstract)`	What the runtime MUST be able to do **without** a vendor CLI
`Inputs / outputs`	Substrate artifacts
`Gate`	A quality checkpoint or a check driven by the runner

2. MVR ↔ Agent Actions

clause_id	Agent action (abstract)	Inputs	Outputs	Gate
MVR-1	Block reverse-engineering of SR/SPEC from code without bug-fix justification; the Source of Truth = the requirements hierarchy	code diff, SR/SPEC	audit record / blocked promotion	—
MVR-2	Verify that the substrate declares V1–V6 in the manifest; run capability checks	`RENAR-CONFORMANCE.yaml`	pass/fail report	—
MVR-3	Reactive, stage-independent ADAPT: create an ADAPT (0..N per TZ) only on findings from adversarial review; no findings → source.tz-section + adversarial-review-ref; delta-TZ → the same reactive pattern; supersession via superseded	TZ, adversarial verdict, ADAPT draft	ADAPT approved / verdict evidence	QG-ADAPT-approve
MVR-4	Reject a SPEC whose type ∉ the closed list	SPEC frontmatter	validation error	QG-0
MVR-5	Ensure a pos/neg pair of TCs for every normative statement	SR/SPEC, TC set	paired TC	QG-2

MVR-6	Reject custom gate types; require QG-0..2	project config	manifest	—
MVR-7	Require a signed `RENAR-CONFORMANCE.yaml` with `senar-version`	manifest	conformance claim	—

3. MVR ↔ Mandatory Clauses §13.3 Bijection

The full MVR ↔ §13.3 bijection is in `08-conformance-self-assessment.md §1`. The agent MUST NOT consider a project conformant if any `mandatory-clauses-confirmed` field = false.

4. Gate Runner Contract (abstract)

Gate	Pre (agent checks)	Post (agent records)
QG-0	Schema valid; parent links; source resolved: `source.adapt` to an ADAPT in `approved` **or** `source.tz-section` + `source.adversarial-review-ref` to an AR with status `issued` (§7.4.1)	`approved` transition + audit-trail
QG-1	TC only (§10.3.2): `automation.status: automated` (valid `automation.location`) or `manual-pending`; implementation pinned to the artifact version (V5); pos/neg pairing; mandatory TC body sections (§9.4)	TC `draft` → `ready` + transition-log record
QG-2	All `verified-by` TC `passing`; pos/neg; `last-run.requirement-version` match	artifact → `verified`

Decision trees: standard/09 §9.1.1, standard/10 §10.1.1.

5. Substrate-Neutrality Rule for Implementers

The runtime MUST map abstract actions onto substrate-native primitives via `RENAR-CONFORMANCE.yaml#substrate-capabilities` (§3.7). Vendor-specific commands MUST NOT be the only way to satisfy a mandatory clause.

6. Coverage Status (v1.0)

Area	Index entries	Profile rows	Note
MVR	7/7	7/7	complete
§13.3 mandatory	8/8	MVR-1..MVR-6; §13.3.7 / §13.3.8 are outside any MVR	no bijection: MVR-7 maps to §13.4
QG-0..2	3/3	3/3	QG-3/4 deferred optional

Mandatory clauses by chapter	partial	—	expand in later pass

Mapping to External Standards

> ***Informative.*** *Elaboration of standard/14 §14.5. Does not alter the mandatory clauses.*

1. SAFe 6.0

A scaled-Agile framework. RENAR borrows the hierarchy mapping (§4.13.1): Portfolio Epic → BR, Feature → SR, Story → TR. Built-in quality (TC up to approved), WSJF for the `priority` field.

2. Spec-Driven Development

An industry term from 2024–2025: under AI acceleration, the correctness of the specification becomes critical. RENAR formalizes the Source-of-Truth inversion (§2.3.1) — a normative structure not tied to a single vendor tool.

3. EARS (Mavin et al., 2009)

Controlled-natural-language templates for SR (§6.6.3) and Pass/Fail in TC.

4. BDD / Gherkin / Specification by Example

Prior art for a full-fledged TC (§9): Given/When/Then, pos/neg as a blocking gate, version pin V5.

5. NIST AI RMF 1.0

Govern / Map / Measure / Manage — a functional mapping to RENAR roles (§5), metrics (§12), and the deprecate lifecycle.

6. IEEE 830-1998 (deprecated)

Withdrawn in favor of ISO/IEC/IEEE 29148. The normative successor is §14.4.2.

7. BABOK v3

Gap: elicitation is out of scope (the TZ is already fixed); Solution Evaluation — partially QG-4 and §12.5.

8. PMBOK 7

"Principles over processes" — RENAR standardizes **what**, not **how** (§2.5).

9. ISTQB Foundation

A testing vocabulary; compatible with RENAR terminologically, but there is no value-for-value correspondence: TC types are RENAR's own closed list `tc-type` (§9.5): `business` / `ux` / `system` / `contract` / `eval` / `security`; test levels are expressed by the separate `level` field (§9.3): `system` / `subsystem` / `module`.

10. CMMI v2.0

Prior art for the RENAR-1..5 levels (§11); CMMI's process-heavy artifacts are not the baseline level.

11. ISO/IEC 42001:2023 (AIMS)

Organizational governance; RENAR provides the evidence base for the requirements slice (ai-provenance, manifest).

12. ISO/IEC 25059:2023

An extension of SQuaRE for AI; a vocabulary for the `quality-characteristic` of AI components.

13. EU AI Act (Reg. 2024/1689)

The `ai-act.risk-class` field in BR; legal conformance is outside RENAR.

14. SysML / MBSE

Prior art for "requirements as a graph" — reference/05; RENAR derives the graph from textual artifacts.

Reference RENAR 1.0 — renar.tech

Document templates BR / SR / TR / TC / AR / ACTZ / AT

> ***Status:*** *informative. This is a* ***possible implementation*** *of the normative schemas — organizations may adapt the skeletons to their substrate and editorial conventions. The normative text these templates merely illustrate:* `standard/06` *(BR / SR / TR),* `standard/07 §7.4.6` *(AR),* `standard/07 §7.13` *(ACTZ) and* `standard/09` *(TC, AT). Where a skeleton and a chapter disagree,* ***the chapter prevails****.*
>
> ***The closed list is untouched:*** *this appendix provides skeletons for already-normative types; new artifact or SPEC types are added only through the formal standard-change procedure (chapter 13). AR is an evidence record, not a requirement type, and belongs to no closed list (§1.7.5).*

12.1 Status and how to use it

Each section below carries two skeletons: a `yaml` block with the frontmatter (with per-line comments on field obligation) and a `markdown` block with the body skeleton. To get a ready artifact file, concatenate both parts into a single substrate document: frontmatter on top, body next.

How the skeletons map to the normative schemas:

Artifact	frontmatter (normative)	Body sections (normative)
BR	§6.5.2	§6.5.3
SR	§6.6.2	§6.6.3
TR	§6.7.2	§6.7.3
TC	§9.3	§9.4
AR	§7.4.6.2	§7.4.6.2
ACTZ	§7.13.3	§7.13.3, §7.13.5
AT	§9.19.3	§9.19.3

Conventions used in the skeletons: `<...>` is a placeholder to replace; `NN` is a sequential number within the scope; the comment `# conditional` marks a field whose obligation depends on a condition (explained inline); `# auto` marks a field maintained by the substrate/runner, not filled in by hand.

12.2 BR template

A BR captures a business need at the system or subsystem level; technical detail is prohibited in a BR (§6.5.1). Frontmatter per §6.5.2; body per §6.5.3.

```yaml
---
id: BR-NN                           # immutable; NN sequential within the scope
title: "<short, descriptive>"
type: BR
slug: "<kebab-case>"                # auto-derived

# === Scope (mandatory) ===
level: system | subsystem           # a BR at module level is prohibited (§6.4)
scope:
  system: "<system-id>"
  subsystem: "<subsystem-id>"       # null if level=system

# === Lifecycle (mandatory) ===
status: draft | approved | verified | deprecated
owner: "<role / responsible person>"

# === Source: provenance (see §7.4.1) ===
source:
  tz-section: "§N.N"                # always mandatory — primary provenance from
the TZ
  adapt: ADAPT-NNN                  # conditional: present if an ADAPT was
created
  adapt-section: "Forward §N"       # mandatory if adapt is set
  adversarial-review-ref: AR-NNN    # conditional: if adapt is absent — the AR
carrying the "no findings" verdict (§7.4.1.2)

# === Cross-level link subsystem BR → system BR (see §6.8.2) ===
implements:                         # array; not a parent-edge, a separate edge
type
  - id: BR-NN                       # id of the parent system BR
    scope:
      system: "<system-id>"
    rationale: "<short>"            # optional; reference to an ADAPT section if
present

# === Relationship graph (substrate-managed) ===
children: []                        # auto: SR whose parent.id = this BR
implemented-by: []                  # auto: subsystem BR referencing it via
implements[]
verified-by: []                     # auto: TC verifying through SR

# === AI provenance (mandatory on RENAR-4+; schema — §4.10.1) ===
ai-provenance:
  generated-by: "<vendor>-<model>@<date>"
  generated-at: "<ISO-8601>"
  human-edits: boolean

# === Replacement (mandatory if applicable) ===
replaces: "<old-id>"
replaced-by: "<new-id>"
```

```
deprecated-date: "<ISO date>"
---
```

```
## Need

Who (role), what (action), why (business goal) — one sentence.

## Success criteria

1. <Measurable outcome, independently verifiable.>
2. <…> (3–7 items in total.)

## Context

Where the requirement came from (with a reference to an ADAPT section if
present);
what alternatives were considered.

## Constraints

<Optional: business constraints — budget, deadlines, regulation. Technical
constraints do not belong here — the SPEC types and SR exist for those.>
```

The `source.tz-section` field is always present; `source.adapt` is omitted when the adversarial review returns a "no findings" verdict — then `source.adversarial-review-ref` is mandatory (§7.4.1).

> ***Where the "requirements" live in a BR.*** *The BR template deliberately has no section with "the system shall …" statements: in RENAR those statements are SR (§6.6), derived from this BR. Mixing them into the BR blurs the "business need ↔ system requirement" boundary and produces "technical BRs" indistinguishable from SR (§6.4, §6.5.1). The BR's own verifiable, enumerable content is carried by the* ***Success criteria*** *section: 3–7 measurable, independently checkable outcomes — these are the business requirements in verifiable form. Decomposing BR → SR turns each criterion into one or more normative SR ("the system shall …" form per §6.6.3).*

12.3 SR template

An SR captures what the system does (observable behavior and constraints); table, framework, and data-structure names are the responsibility of SPEC (§6.6.1). Frontmatter per §6.6.2; body per §6.6.3.

```
---
id: SR-NN                              # immutable
title: "<short, descriptive>"
type: SR
slug: "<kebab-case>"

# === Scope (mandatory) ===
```

```yaml
level: system | subsystem | module
scope:
  system: "<system-id>"
  subsystem: "<subsystem-id>"        # null if level=system
  module: "<module-id>"              # null if level ≠ module

# === Lifecycle (mandatory) ===
status: draft | approved | verified | deprecated
owner: "<role / responsible person>"

# === Parent (mandatory) ===
parent:
  id: BR-NN                          # single parent

# === Source: provenance (see §7.4.1; same rules as BR) ===
source:
  tz-section: "§N.N"                 # always mandatory
  adapt: ADAPT-NNN                   # conditional
  adapt-section: "Forward §N"        # mandatory if adapt is set
  adversarial-review-ref: AR-NNN     # mandatory if adapt is omitted — the AR
(§7.4.6)

# === Quality characteristic (conditional; §6.6.2) ===
# mandatory for a non-functional SR; the value comes from the ISO/IEC 25010:2023
list.
# Omitted for a functional SR.
quality-characteristic: functional-suitability | performance-efficiency |
compatibility |
                        interaction-capability | reliability | security |
maintainability |
                        flexibility | safety

# === Relationship graph ===
constrained-by:                      # typed edges to SPEC (chapter 8)
  - SPEC-UI-NN
  - SPEC-API-NN
  - SPEC-DATA-NN
children: []                         # auto: TR whose parent.id = this SR
verified-by: []                      # auto: TC verifying the SR

# === AI provenance (mandatory on RENAR-4+) ===
ai-provenance:
  generated-by: "<vendor>-<model>@<date>"
  human-edits: boolean

# === Replacement (mandatory if applicable) ===
replaces: "<old-id>"
replaced-by: "<new-id>"
deprecated-date: "<ISO date>"
---
```

```
## Requirement

One sentence in normative form: "The system MUST …" (modality per the convention
of §0.5).

## Behavior

A detailed description of observable behavior; functional scenarios.

## Constraints

<Mandatory if applicable: non-functional constraints — performance, security.
Full constraints are pushed into SPEC via constrained-by[].>

## Link to SPEC

<Mandatory if constrained-by[] is present: which aspects of behavior are governed
by which SPEC.>
```

`parent.id` is the single BR (a parent tree); `constrained-by[]` is a graph of references to SPEC of any type and in any number (§6.6.2).

12.4 TR template

A TR is the atomic unit of implementer work: exactly what to build within a single SR (§6.7.1). Frontmatter per §6.7.2; body per §6.7.3.

```
---
id: TR-NN                           # immutable
title: "<short, descriptive>"
type: TR
slug: "<kebab-case>"

# === Scope (mandatory) ===
level: system | subsystem | module   # system — rare, cross-subsystem tasks
scope:
  system: "<system-id>"
  subsystem: "<subsystem-id>"        # null if level=system
  module: "<module-id>"              # null if level ≠ module

# === Lifecycle (mandatory) ===
status: draft | approved | done | obsolete
owner: "<assignee role / agent>"

# === Parent (mandatory) ===
parent:
  id: SR-NN                          # single parent

# === Source: traceability chain (inherited from parent SR, §6.7.5) ===
```

```yaml
source:
  adapt: ADAPT-NNN                  # auto: inherited from parent SR; may be
absent
  sr-version: "<version-ref>"       # pinning to the SR version (substrate
capability V5)

# === Relationship graph ===
implements-spec:                    # typed edges to SPEC
  - SPEC-API-NN
  - SPEC-UI-NN
verified-by: []                     # auto: TC verifying through SR

# === Goal and acceptance criteria ===
goal: "<one-sentence outcome>"
acceptance-criteria:
  - "<numbered, falsifiable, unambiguous>"
  - "<…>"

# === AI provenance (mandatory on RENAR-4+) ===
ai-provenance:
  generated-by: "<vendor>-<model>@<date>"
  human-edits: boolean
---
```

```markdown
## Goal

One paragraph; the outcome the TR makes observable.

## Acceptance Criteria

1. <Falsifiable criterion; covers a positive scenario.>
2. <Falsifiable criterion; covers a negative scenario / boundary.>

## Scope

What is in and what is **not** in the TR (per SENAR Rule 2).

## References

<Mandatory if applicable: to the SPEC in implements-spec[] and the sections of
the
parent SR.>
```

The section names `Goal`, `Acceptance Criteria`, `Scope` are canonical per §6.7.3. The TR implementer works within the SR / SPEC and does not reach the ADAPT directly (§6.7.5).

12.5 TC template

A TC verifies a normative assertion of a BR / SR / SPEC; the common frontmatter is per §9.3, the body per §9.4. Type-specific fields (`judge` , `baseline` for `ux` / `eval`) are added on top per §9.6.

```
---
# === Identity (mandatory) ===
id: TC-NN                              # immutable; NN sequential within the scope
title: "<short, descriptive>"
type: TC
slug: "<kebab-case>"

# === Classification (mandatory) ===
tc-type: business | ux | system | contract | eval | security   # business — the
canonical name (§9.5)
negative: boolean                      # true for the paired negative TC

# === Scope (mandatory) ===
level: system | subsystem | module
scope:
  system: "<system-id>"
  subsystem: "<subsystem-id>"          # null if level=system
  module: "<module-id>"                # null if level ≠ module

# === Lifecycle (mandatory) ===
status: draft | ready | passing | failing | obsolete

# === Verification target (mandatory; at least one) ===
verifies:
  - id: SR-NN | BR-NN | SPEC-<TYPE>-NN
    requirement-version: "<version-ref>"   # artifact version pinning (V5)

# === Pair link (mandatory if negative=false and a paired TC exists) ===
paired-with:
  - TC-NN

# === Task binding (optional; §9.19.7) ===
verifies-tr: TR-NN                     # the task to whose scope the test is
narrowed
verifies-claims: []                    # the subset of the parent SR's claims
covered within that TR

# === Bench and data (conditional; §9.3) ===
environment-ref: SPEC-TEST-NN          # mandatory if automation.kind: dynamic —
the bench and dataset
                                       # the TC is valid against. Does not apply to
static checks
                                       # (doc-lint, structural TC): the analyzer
works over artifacts

# === Automation (mandatory) ===
automation:
  status: automated | manual-pending
  kind: dynamic | static                                # mandatory; static — the
runner is a static analyser
```

```
  location: "<substrate pointer to implementation>"  # mandatory if automated
  manual-pending-until: "<ISO date>"                  # mandatory if manual-
pending
  manual-pending-reason: "<text>"                     # mandatory if manual-
pending

# === Red history (§9.18.2; the condition for counting a TC as evidence) ===
red-history:                          # auto: maintained by the substrate from run
facts
  fixing-run:                         # the fixing run before implementation; MUST
be red
    date: "<ISO-datetime>"
    result: fail
  green-transition:                   # the recorded red → green transition
    date: "<ISO-datetime>"
  inherited-from: null                # conditional: TC-NN when behaviour did not
change (refactoring, migration)
  not-applicable-reason: null         # conditional: implementation-originated —
then a killed mutant is mandatory
  mutation-check:                     # mandatory when not-applicable-reason is
set
    mutants-killed: 0                 # >= 1, otherwise the TC is not evidence

# === Execution (mandatory for tc-type: ux | eval) ===
judge:
  vendor: "<provider>"                # mandatory; judge isolation — P7
  model: "<model-id>"
baseline:                             # mandatory for ux | eval
  artifact: "<substrate pointer>"
  perceptual-diff-threshold: float    # for ux
  metric-thresholds: {}               # for eval

# === Last run (runner-managed; not filled by the author) ===
last-run:                             # auto
  date: "<ISO-datetime>"
  result: pass | fail | skipped | n/a
  runner-id: "<runner-name@version>"

# === AI provenance (mandatory on RENAR-4+) ===
ai-provenance:
  generated-by: "<vendor>-<model>@<date>"
  human-edits: boolean
---
```

Context

Which clause of the verified artifact the TC references; a quote or paraphrase of the assertion.

Preconditions

The system and data state required for the run; provided by the seed mechanism.

```
## Steps

Runner actions. For tc-type: ux — intentions, not selectors (§9.6.1).

## Pass criterion

Binary, observable, reproducible (§9.11).

## Fail criterion

A list of observable signs of a violation (not the negation of the Pass
criterion):
leaks, side effects, race conditions.

## Postconditions

The state expected after the run; the cleanup mechanism.

## Out of scope

What is **deliberately** not checked, naming the paired TC where it is covered.
```

The `environment-ref` field is mandatory for dynamic TCs (`automation.kind: dynamic`): "the test passed" only means something against a specific bench and specific data (§9.3). It does not apply to static checks (the doc-lint, the structural TC) — the analyzer works over artifacts. The reference points to a SPEC-TEST (§8.5.10); changing the bench or the dataset increments its version and invalidates `verified` on the dependent TCs — precisely, without touching anything else.

The headings `## Pass criterion` and `## Fail criterion` are fixed: the change-of-criteria control hook detects them (§10.11.3) — they may not be renamed locally. The "Out of scope" section is mandatory: its absence blocks the TC transition to `ready` (§9.4).

12.6 AR template — the adversarial-review record

AR captures the fact and the outcome of the mandatory adversarial review (§7.4.6). It is an **evidence record**, not a requirements artifact: it has no parents, no children and no test cases. It is issued in both outcomes of the review — whether an ADAPT is created or not.

```
---
id: AR-NNN                          # immutable; NNN is sequential within the
parent TZ
type: AR
tz-ref: TZ-YYYY-NNN                 # mandatory; the (delta-)TZ under review
trigger-stage: import-tz            # mandatory: import-tz | decompose-br |
decompose-sr | spec | tc

# === Reviewer (mandatory; isolation §7.10.2) ===
reviewer:
  vendor: "<provider>"
```

```
  model: "<model-id>"                # MUST differ from the primary agent's model

# === Verdict (mandatory) ===
verdict: no-findings                 # no-findings | findings-present
produces-adapt: []                   # conditional: non-empty when findings-
present; empty when no-findings

# === Lifecycle (mandatory) ===
status: issued                       # draft | issued | superseded
superseded-by: null                  # conditional: AR-NNN when status=superseded

# === Signature (mandatory for issued; substrate capability V6) ===
signature:
  author: "<reviewer-id>"
  timestamp: "<ISO-8601>"
---
```

Body on `verdict: no-findings`:

```
## Justification of unambiguity

Why converting the reviewed TZ sections into requirements produces no gap: how
each risk
(terms, scope boundary, tacit assumptions) is closed.

## Review coverage

The TZ sections included in the review, and the derivation stage at which it was
conducted.
```

Body on `verdict: findings-present`:

```
## Review coverage

The TZ sections included in the review, and the derivation stage.

## Findings produced

Pointers to the `B-NNN` records in the ADAPT that was produced — **without
duplicating** their
content: the findings themselves live in the ADAPT (§7.4.4).
```

An issued AR is immutable. If a repeat review at the same stage issues a new verdict, a new AR is issued and the previous one moves to `superseded` with `superseded-by` filled in. Referencing an AR in status `draft` or `superseded` from `source.adversarial-review-ref` is prohibited — the gate reports a fatal error (§10.11.1).

12.7 ACTZ template — the TZ clarification protocol

ACTZ is an artifact of the contractual contour: a batch of questions, proposals and decisions put to the client and signed by both parties (§7.13). Decisions are worded in the language of obligations ("the button is named X"), not of interpretation. The frontmatter is per §7.13.3; TZ annexes are per §7.13.5.

```yaml
---
id: ACTZ-NNN                          # immutable; sequential within the parent TZ
title: "TZ Clarification Protocol No. N"
type: ACTZ
tz-ref: TZ-YYYY-NNN                   # mandatory; the TZ the protocol belongs to

# === Findings closed (conditional) ===
resolves:                             # B-NNN entries in an ADAPT that the
protocol closes
  - id: B-NNN                         # absent on a client-initiated protocol
(§7.13.2)
    adapt: ADAPT-NNN

# === Decisions (mandatory; non-empty list) ===
decisions:
  - number: "§M"                      # stable clause number; the target of
decided-in
    statement: "<decision in the language of obligations>"
    tz-section: "§N.N"                # the TZ section being clarified

# === TZ annexes (conditional; §7.13.5) ===
annexes:
  - name: "<annex>"
    version: "<new version>"
    document-ref: "<substrate link>"

# === Lifecycle (mandatory) ===
status: draft                         # draft | sent | signed | superseded
superseded-by: null                   # conditional: ACTZ-NNN when
status=superseded

# === Signatures (mandatory for signed; substrate capability V6) ===
client-signature:
  signed-by: "<name>"
  role: "<role of the authorised representative>"
  organization: "<client organization>"
  signed-at: "<ISO-datetime>"
vendor-signature:
  signed-by: "<name>"
  role: "<role>"
  signed-at: "<ISO-datetime>"
---
```

```
## Subject of the clarification

Which TZ sections are clarified and why the question arose — with a reference to
the TZ
section (and to the `B-NNN` finding in the ADAPT where one exists).

## Decisions

1. **§1.** <The decision in the language of obligations; a checkable wording.>
2. **§2.** <…>

Clause numbering is stable: `B-NNN.decided-in` and `AT.verifies[]` point at it.

## Annexes

<Mandatory where applicable: the new versions of the TZ annexes approved by this
protocol
(§7.13.5). The previous version remains immutable.>

## Signatures

A two-sided signature: the client (or an authorised representative) and the
vendor.
```

A signed ACTZ **MUST NOT** be edited: a correction is made only by a new protocol that supersedes the earlier decision (`superseded-by`). Referencing an ACTZ in status `draft` from `decided-in` is prohibited (§7.13.4). The effective TZ = the initial TZ with its annexes plus every signed ACTZ (§7.14).

12.8 AT template — the acceptance test of the contractual contour

An AT is derived **exclusively** from the effective TZ and checks conformance to the contract, not to the interpretation (§9.19). An AT is created by an isolated agent: only the effective TZ is given as input; access to ADAPT, BR / SR / SPEC, TC and code is prohibited, and a breach of isolation is fatal. The frontmatter and body are per §9.19.3.

```
---
id: AT-NN                           # immutable
title: "<short, descriptive>"
type: AT
negative: boolean                   # mandatory; pos/neg pairing — as for TC
(§9.7)

# === Verification target (mandatory; contractual references only) ===
verifies:
  - "TZ §N"                         # a section of the effective TZ
  - "ACTZ-NNN §M"                   # a clause of a signed protocol
  # a reference to BR / SR / SPEC / TC is fatal (§9.19.2)

tz-version: "<effective TZ revision>"  # mandatory; the revision the AT was
```

```
derived from

# === Provenance of the isolated agent (mandatory) ===
generator:
  vendor: "<provider>"
  model: "<model-id>"                # MUST differ from the primary agent's model
  internal-contour-access: false     # mandatory; confirmation of no access to
the internal contour
  generated-at: "<ISO-8601>"

# === Lifecycle (mandatory) ===
status: draft                        # draft | ready | passing | failing |
obsolete

# === Trial environment and dataset (mandatory; §9.19.3) ===
environment-ref: SPEC-TEST-NN        # set by the Architect or the runner AFTER
generation:
                                     # the isolated agent does not see internal
artifacts (§9.19.2)

# === Automation (mandatory) ===
automation:
  status: automated | manual-pending
  kind: dynamic | static
  location: "<substrate pointer to implementation>"

# === Last run (runner-managed; not filled by the author) ===
last-run:                            # auto
  date: "<ISO-datetime>"
  result: pass | fail | skipped | n/a
  runner-id: "<runner-name@version>"
  tz-version: "<effective TZ revision at the time of the run>"
---
```

```
## The effective-TZ clause

> "<The verbatim quotation of the clause under test — a TZ section or a clause of
a
> signed ACTZ.>"

This section is mandatory (`tz_text`, §9.19.3): at acceptance the clause of the
contract
itself is produced, not a paraphrase.

## Preconditions

The system and data state required for the run.

## Steps

Runner actions — worded from the client's standpoint, without relying on the
internal
```

```
construction of the system.

## Pass criterion

Binary, observable, reproducible; derived from the quoted clause.

## Fail criterion

A list of observable signs of non-conformance to the contract.

## Out of scope

What is **deliberately** not checked, naming the paired AT where it is covered.
```

ATs **are regenerated before every trial** from the current revision of the effective TZ: an outdated trial programme MUST NOT be admitted to the run (§9.19.4). The product is not submitted for hand-over until every AT is in status `passing` (§10.4.3).

12.9 SPEC templates — deferred

Skeletons for the SPEC types (chapter 8) are **deliberately not included** in this appendix. The partner review flagged the question of SPEC skeletons as open: the set of mandatory fields differs across SPEC types (UI, API, DATA, AI, SEC, INT) far more than across BR / SR / TR / TC, and fixing a skeleton prematurely risks reading as normative.

Draft sketches of the SPEC skeletons are kept in the repository — `research/17-specification-schema-and-templates.md` (§5; an internal draft, not published to the site) — pending a separate decision. When that decision is made, the SPEC skeletons are added here as a new section — without changing the closed list of SPEC types, which is already normative in `standard/08 §8.2.2` and §8.3.

The closed list now holds **eleven** types: the nine structural ones plus `SPEC-TEST` (benches and data — how conformance is proven) and `SPEC-DOC` (the composition of the delivered documentation). The deferral of skeletons covers them too: the field schemas of both types are given in `reference/02` (sections 6.3 and 6.4); document skeletons are not.

12.10 Filling placeholders and common mistakes

Placeholder	What to replace it with	Common mistake
`BR-NN` / `SR-NN` / `TR-NN` / `TC-NN` / `AT-NN`	A sequential ID within the scope (the substrate assigns it on creation)	Changing the ID after publication — it is immutable
`ACTZ-NNN` / `decisions[].number`	The protocol's sequential number and the stable clause number of a decision	Renumbering the clauses of a signed protocol — `decided-in` and `AT.verifies[]` point at them

`AT.verifies[]`	Only `TZ §N` and `ACTZ-NNN §M`	Referencing an SR / SPEC / TC — a breach of isolation; the gate reports a fatal error
`<system-id> / <subsystem-id> / <module-id>`	Identifiers from the project's system registry	Filling `subsystem` when `level=system` (it must be `null`)
`source.tz-section`	The section of the source TZ — always present	Omitting it on the assumption that a reference to the ADAPT is enough
`constrained-by[] / implements-spec[]`	IDs of existing SPEC	Naming a SPEC type outside the closed list
`# auto` fields (`children`, `verified-by`, `last-run`)	Nothing — the substrate / runner maintains them	Filling them by hand and diverging from the relationship graph

Placeholders like `BR-NN` and empty `<...>` are an unfilled skeleton, not a valid artifact: run such files through checks only after substituting real values, otherwise the substrate validators will rightly reject the template IDs.

← Back to the reference overview

Recommended TZ form

> ***Status:*** *informative. This is a* ***recommendation****, not a norm. RENAR conformance does not depend on the form of the incoming TZ in any clause: the standard accepts the TZ* ***as given*** *— including a poorly structured one — and normalizes not the writing of the TZ but the work with it (* `standard/01 §1.3` *: RENAR does not normalize the TZ authoring process on the client's side). The only normative mechanism for working with a TZ of any quality is the ADAPT contour (* `standard/07` *).*
>
> *This appendix creates no obligations and extends no closed list (§1.7.5).*

13.1 Status and boundaries

The TZ is a document external to the standard: the client writes it or brings it, it lives in the contractual contour, and it follows the vendor's business practice. The whole machinery of the standard — forward adaptation, backward findings, ACTZ — is built precisely on the assumption that an incoming TZ may be anything at all. Were the standard to require a TZ form, it would stop accepting other parties' TZs and would lose its own input condition.

Hence three hard boundaries, upheld across the corpus:

1. **Conformance does not depend on the TZ form.** The conformance chapter (`standard/13`) does not mention the form of the incoming TZ in any clause. An organization whose TZ has an arbitrary structure may claim full conformance.
2. **Substrate checks do not touch the TZ structure.** No automated check parses a TZ into sections or requires that any be present. Requiring a TZ structure through an automated check would violate the boundary of §1.3.
3. **The TZ does not become a requirements registry.** Duplicating BR / SR into the TZ is an explicit anti-recommendation (§13.5).

The only admissible argument in favour of the form is an observation, not an obligation: a TZ written to this form produces fewer backward findings, shortens the clarification rounds, and makes the derivation of acceptance tests reliable (§13.6). The choice remains with the vendor and the client.

13.2 The basis of the form: an inversion of the finding typology

The form is recommended not out of one vendor's habit but on a basis internal to the standard. The seven categories of backward findings (§7.4.4) are, in essence, a catalogue of typical TZ defects. Therefore the **recommended form is an inversion of the finding typology**: each section exists not for its own sake but as prophylaxis against a specific category of defect.

Section of the form	Category of findings it forecloses
Out of scope	`scope` — an unclear boundary of work
User roles (explicit authorization)	`hidden-assumption` — an assumption that may be wrong
Integrations (constraints, risks)	`feasibility` , `regulatory`

Definitions	`terminology` — direct prophylaxis
Key data entities	`terminology` (entities), `hidden-assumption`
"Given — when — then" criteria on every functional requirement	`gap` — the TZ is silent about something without which implementation is impossible
Use-case scenarios with pre- and postconditions	`contradiction` — scenarios collide requirements against one another
Accompanying artifacts (a table)	addressability of annexes: each annex becomes a unit that can be referenced

The table also works in reverse — as an instrument for finding holes in the form itself. It is precisely this table that exposed the fact that the `terminology` category, until the "Definitions" section appeared, was the only one without a prophylactic section.

13.3 Sections of the form

Eleven sections. The numbering is internal to the TZ; the stable identifiers of clauses (`UC-NNN` , `FR-NNN` , `NFR-NNN`) are assigned by the client and do not change afterwards: acceptance tests reference them (§9.19.3).

#	Section	Content
1	General description	One paragraph: what this is, for whom, how it works
2	**Definitions**	The client's terms with fixed meanings (§13.4)
3	User roles	A table of roles; authorization is stated explicitly — including an explicit "not provided"
4	Use-case scenarios	`UC-NNN` : participant, precondition, steps, postcondition
5	Functional requirements	`FR-NNN` : priority and "given — when — then" acceptance criteria, including at least one negative variant with invalid data
6	Non-functional requirements	`NFR-NNN` : only what is explicitly constrained or significant
7	Integrations	A table per external system: address, type, constraints, data in and out, risks
8	Key data entities	A vocabulary of the problem domain — not a database schema
9	Accompanying artifacts	A table of annexes: type, coverage, location, status
10	Out of scope	An explicit list of what is not included
11	Project acceptance criteria	Conditions of acceptance, the reference scenario

Sections 4 and 5 are what the acceptance-test programme is derived from (`reference/14`). Section 11 is its embryo: from the outset the client sees how the handover will proceed.

13.4 The "Definitions" section — second, and why exactly there

The section is placed **second**, immediately after the general description and before the first use of terms in scenarios and requirements. This follows GOST and ISO practice, where terms and definitions come at the start of a document.

The content is the client's terms that admit a second reading: roles and their synonyms, action verbs ("post", "approve", "export"), statuses, units of measure, the organization's abbreviations. The form is a table: "term — meaning — synonyms, or what the term is not".

Term	Meaning	Synonyms / what it is not
Post a document	Commit a document to the ledger such that it affects balances	Not "save": a draft does not change balances

Two recommendations for filling it in:

- include only terms with a possible second reading — the section is not a substitute for a dictionary of ordinary words;
- define a term with a self-contained formulation. Definition by usage ("as in section 4") is inadmissible: it does not remove the ambiguity, it hides it.

The section is a direct input for term mapping in ADAPT: a term fixed by the client produces no finding of the `terminology` category, and the whole terminological layer need not be reconstructed through rounds of clarification, when the client could have fixed their own terms up front.

13.5 What a TZ is not

The TZ remains a document in the language of the client and their problem domain. The form structures the client's language — it does not turn the TZ into a requirements registry.

Duplicating BR / SR into the TZ is **not recommended**. The requirements of the standard live in the internal contour and are derived from the TZ through interpretation (§7.12); copied into the TZ, they diverge from the original at the first edit and produce two sources of truth instead of one. Decoupling the contours is not a formality but a condition for traceability to work.

The distance between the language of the TZ and the language of the standard is variable and normal (§7.12.1). The form shortens that distance; it does not abolish it.

13.6 The observed effect

The effect is recorded as an observation — neither a promise nor an obligation:

- fewer backward findings — each section of the form forecloses its own category in advance (§13.2);
- a shorter path to requirements — fewer ACTZ clarification rounds (§7.13);
- a more reliable derivation of acceptance tests: stable `UC-NNN` and `FR-NNN` identifiers give the `verifies` field its addressing without interpretation, while the mandatory negative variant in the acceptance criteria yields positive/negative pairing of acceptance tests almost for free (§9.19).

A vendor with their own TZ loses only speed, not conformance.

13.7 Relation to other documents

Document	Relation
`standard/07 §7.4.4`	The seven categories of backward findings — the basis of the form
`standard/07 §7.13`	ACTZ — the TZ clarification protocol
`standard/07 §7.14`	The effective TZ — the benchmark for acceptance
`standard/09 §9.19`	AT — the acceptance test of the contractual contour
`reference/14`	Recommended form of the acceptance-test programme
`reference/12`	Templates for internal artifacts (BR / SR / TR / TC / AR / ACTZ / AT)

← Back to the reference

Recommended form of the acceptance-test programme

> ***Status:*** *informative. The recommendation concerns the* ***human-readable document produced for the client****. The structure of the AT artifact itself is normative and is set by* `standard/09 §9.19` *; where a skeleton and a chapter disagree,* ***the chapter prevails****.*
>
> *This appendix creates no obligations and extends no closed list (§1.7.5).*

14.1 What this document is

The acceptance-test programme is the scenario document produced for the client at handover. It is the **client-facing projection of the set of ATs**: the machine executes the acceptance tests, the human reads the programme. That split continues a general principle of the standard: the client sees the human-readable, the machine works with the canonical.

The programme is **derived from the effective TZ** (§7.14) — the initial TZ with its annexes plus every signed ACTZ — and not from the initial TZ. The contract lives incrementally, and a system cannot be produced against a stale revision: ATs are regenerated before every trial from the revision in force (§9.19.4), and the programme is regenerated after them.

The programme is generated, not hand-written: everything in it already exists in the ATs.

14.2 Fields of the programme

Field	Content	Source in the standard
`id`	The stable identifier of the contract clause (`UC-NNN` , `FR-NNN`) or of an ACTZ clause. Not invented — taken from the document	`reference/13 §13.3`
`source`	Where the clause comes from: `TZ §N` or `ACTZ-NNN §M` , including subsystem TZs	`verifies[]` , §9.19.3
`tz_text`	The **verbatim quotation** of the effective-TZ or ACTZ clause	a mandatory body section of an AT, §9.19.3
`steps`	Verification steps as active commands: "open…", "click…", "confirm…"	AT body
`expected`	The expected result in one sentence	AT body

An example of a single programme entry:

```
id: FR-014
source: "TZ §5.14"
tz_text: >
```

```
  The system shall reject posting of a document when the sum of the
  line items diverges from the sum stated in the header.
steps:
  - "Open a document whose header sum diverges from its line items."
  - "Click «Post»."
  - "Confirm the document is not posted and the reason for refusal is shown."
expected: "The document remains a draft, and the reason for refusal is stated
explicitly."
```

The verbatim quotation is the key field. The client sees the text of the contract and the verification side by side: what is produced is not a paraphrase of the clause but the clause itself. There is nothing left to argue about regarding "what was meant" — which is why the quotation was taken into the normative part of the AT rather than left as a recommendation.

14.3 Coverage completeness

Every contract clause enters the programme — with no omissions and with no merging of several clauses into one. Completeness is counted over the sections of the effective TZ and the clauses of the signed ACTZ; the machine-side counterpart of this requirement is the `ACCEPTANCE.md` report (§9.19.6).

Merging clauses looks like a saving, but it removes addressability: the failure of a merged check cannot be attributed to a specific contract clause, and the verdict once again becomes a matter of discussion.

The programme may be agreed with the client through a further ACTZ (§7.13) — the programme then becomes an approved commitment in its own right, and the order of handover is known to both parties in advance.

14.4 The verdict by defect class

The acceptance verdict is best computed **by defect class rather than as a binary**. The mechanism: every failed programme entry is classified, and the decision to sign the acceptance certificate is taken according to quality corridors declared in advance.

A typical gradation has four levels:

Level	Name	Description
1	Critical	The system is inoperable, data is lost or corrupted, security is breached
2	Major	Key functionality is unavailable or works incorrectly, with no workaround
3	Moderate	Secondary functionality is incorrect, a workaround exists
4	Minor	Visual, textual, and other defects that do not affect operation

> ***The gradation is given as an example, not as a norm.*** *The standard establishes **neither** numeric thresholds **nor** the number of levels itself. The quality corridors — which levels block signature of the certificate, how many defects of each level may be carried over — are declared by the **vendor** in the contract or in the manifest, based on their internal quality regulations. A bank and an in-house department will have different thresholds, and both will be right: this is commercial policy, not a property of requirements engineering.*

What is recommended regardless of the thresholds chosen: a defect that blocks signature is recorded with a reference to the contract clause (`id` , `tz_text`), a description of the divergence, and the expected result; non-blocking defects are recorded, and the plan for fixing them is agreed with the client before the certificate is signed.

The value of the mechanism lies in the rules of the game being declared before the trials. Without it, every defect found at acceptance becomes a matter of bargaining.

14.5 The internal gate and contractual acceptance are not the same thing

Quality corridors describe the behaviour of the parties **at acceptance** and do **not** weaken the vendor's internal gate.

Contour	Rule	Who finds the defects
Internal, before handover	Every AT in `passing` status — strictly, with no corridors (§10.4.3)	The vendor
Contractual acceptance	The verdict follows the vendor's quality corridors	The client

A vendor does not produce a system with known failures — otherwise the corridors turn into permission to hand over unfinished work. But at a real acceptance the client finds their own defects, and the classification supplies rules agreed in advance for that situation.

14.6 The defect level as diagnostics

The defect level correlates with the routing of AT and TC failures (§9.19.5):

- a level 1–2 defect with green TCs is almost always an **interpretation error**: the system conforms to ADAPT but not to the contract. It is fixed in ADAPT and ACTZ, not in the code;
- level 3–4 defects more often point to under-specified detail — candidates for a further ACTZ.

14.7 Relation to other documents

Document	Relation
`standard/09 §9.19`	AT — the normative structure of the artifact the programme is derived from
`standard/07 §7.13`	ACTZ — the protocol for clarifying the TZ and agreeing the programme
`standard/07 §7.14`	The effective TZ — the benchmark the trials run against
`standard/10 §10.4`	The internal release gate
`reference/13`	Recommended TZ form — the source of stable clause identifiers

← Back to the reference