

RENAR

Requirements Engineering & Normative Adaptive Regulation — normative
core

RENAR · Standard · Version 1.1 | 19.09.2026

Authors: Vadim Soglaev, Andrey Yumashev

CC BY-SA 4.0 | renar.tech

RENAR Core

Version: 1.1 · **Date:** 2026-09-19 · **Site:** renar.tech **Copyright:** (C) 2026 Vadim Soglaev, Andrey Yumashev.
Licensed under [CC BY-SA 4.0](#).

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), 12 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** — states (draft / approved set version / verified on a build) 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:  
how engineers read the TZ;  
architect's signature"]
    AR -->|«no findings»| ART["BR / SR / SPEC / TR / TC  
(RENAR description)"]
    AD -->|question to the client| ACTZ["ACTZ – the «TZ Clarification Protocol»:  
decisions in the language of commitments;  
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  
derived from the contract"]
    IMPL --> AT
  
```

The diagram shows the **dependencies between artifacts** — what is derived from what and what confirms what — not an order of work. The order is set by the organization's production model; the standard governs only preconditions and rules ([standard/01 §1.2.1](#)).

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 of a description set version, of a task, of an ADAPT), QG-1 (admission of a TC implementation), QG-2 (verification of a set version on a product version) 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

RENAR Core 1.1 — *renar.tech* Copyright (C) 2026 Vadim Soglaev, Andrey Yumashev. Licensed under CC BY-SA 4.0.

00. Introduction

Part of the RENAR Standard v1.1 · ← [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 twelve 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.

The standard does not govern the order of work: it sets preconditions (what **MUST** exist before approval, a transition, a presentation) and the rules linking artifacts; the sequence of actions is the subject of the organization's production model (§1.2.1).

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 / 12 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 a **subset** of the mandatory clauses of §13.3, not a full match: MVR-7 maps onto §13.4, while §13.3.7 and §13.3.8 belong to no MVR.

#	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 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 <code>trigger-stage</code> , and multiplicity is the regular case. When a gap is present, the ADAPT is REQUIRED to be in status <code>approved</code> 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 <code>source.tz-section</code> + <code>source.adversarial-review-ref</code> → AR (§7.4.6). In either outcome the review result is issued as an AR. A delta-TZ follows the same rule.	§7, §13.3.3
MVR-4	Closed list of 12 SPEC types: a SPEC type MUST belong to the closed list <code>SPEC-ARCH / API / DATA / INT / PROC / UI / AI / SEC / OPS / TEST / DOC / UC</code> .	§8.3, §13.3.4
MVR-5	TC pos/neg pairing: every normative statement of a set version that describes observable behaviour MUST be covered by a positive + negative pair of TC norms of the same version — a precondition of version approval. 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 <code>required</code> . QG-3 / QG-4 are <code>declared</code> or <code>absent</code> . Creating new gate types locally is prohibited.	§10.3, §13.3.6
MVR-7	Conformance manifest: a project MUST contain a manifest with both <code>renar-version</code> + <code>senar-version</code> + <code>level</code> (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).

The language of project artifacts is not the subject of this paragraph: the normative sections of BR / SR / SPEC / TC / ADAPT / ACTZ are written by the rules of [chapter 15](#).

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 v1.1 · ← 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 12 specification types (SPEC-ARCH / API / DATA / INT / PROC / UI / AI / SEC / OPS / TEST / DOC / UC)	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** — the state machines of the description set, the task, ADAPT and the TC implementation (§10.7); 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.

The order of work is outside the normative area. The standard sets what **MUST** exist, and in what state, before a version is approved, a transition is made or a result is presented — the preconditions and the rules linking artifacts. The sequence and grouping of actions, the distribution across the roles of a production pipeline, planning — are the subject of the organization's production model (§1.3, item 7). Any order of work under which the preconditions hold conforms to the standard; a sequence given in a guide or an example is one possible order, not a norm. An organization **MAY** add artifacts and rules of its own — that is `declared-stricter` (§1.7.2) as long as the closed lists and the preconditions are not violated.

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 <code>automation.location</code> and <code>last-run</code> (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 and the production process	Agile ceremonies, sprint planning, kanban boards, story-point estimation, the order and grouping of work on artifacts, the distribution across pipeline roles — out of scope; RENAR governs the lifecycles of artifacts and the preconditions of their transitions (§1.2.1), not the order of work (item text refined in v1.1, ADR-027).
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 the <code>ai-provenance</code> block in the composition of §4.10.1 — including <code>prompt-template</code> as a versioned pointer, but not the template's content — 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 <code>guide/03-tool-guide-*.md</code> .
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.

1.4.4 First-party confirmation — a concept as the input

A separate **kind of applicability**, added in v1.1 (§13.9): there is no external client, and nobody imitates that role. Indicators:

1. **The input is a concept**, not a TZ: a written description of the future system, composed by a responsible person of the organization and frozen by them (V6 author and timestamp). The concept takes the place of the TZ as the source (`source.tz-section` points to its section) and carries the same immutability obligations (§7.4.1).
2. **No contractual contour arises**. The ACTZ (§5.5, §7.13) has no subject — there is nobody to put decisions to; the dual signature is **not imitated** by one person. A backward finding that requires a decision on the concept is closed by a revised section of the concept, signed by the same responsible person: `decided-in` points to it (§7.4.5).
3. **The adversarial review is mandatory without exception** (§7.4.1, §13.3.3): it is the only independent check in this kind, so an AR with the verdict "no findings" MUST carry a presentable rationale, the reviewer model is recorded, and the ACR (§12.3.6) is kept mandatorily.
4. **Presentation — against the concept**. The AT release gate (§10.4.3) has no subject: ATs are derived from a contract, and there is none. Readiness is presented by the verification entry of the set version (QG-2, §10.3.3) signed by the responsible person; manual walkthrough records over the scenarios (§9.20) are an additional detection channel and are not part of the proof of readiness.
5. **Labelling of the applicability kind**. The manifest declares `confirmation: first-party` (§13.4.2); the conformance claim is a first-party confirmation in the sense of ISO/IEC 17050-1, not a second-party acceptance.

The kind is orthogonal to the maturity level: it sets no RENAR-N ceiling. The appearance of an identifiable stakeholder with acceptance authority moves the project into the contract-oriented kind (§1.4.1) by withdrawing the declaration; no retrospective re-assessment is required.

Combining roles in this kind is the norm, not a deviation: one person MAY combine roles (§5.5.5); adversarial checking by people is then absent, and this is an accepted trade-off that the standard does not compensate with additional mechanisms but names honestly.

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 a responsible person and a written concept

An internal tool that has neither a responsible person ready to freeze a concept and sign a first-party confirmation (§1.4.4) nor a written concept at all: requirements live in discussions, no "client" is named. The absence of an external client by itself does **not** take a project out of the applicability area — the absence of a person responsible for the description does. Imitating the ACTZ dual signature by one person (`client-signature.signed-by == vendor-signature.signed-by`) remains a violation of §5.5.3: the correct path is the kind of §1.4.4, where the ACTZ has no subject.

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 that has neither a frozen concept nor a responsible person is non-conformant (§13.8). If the product acquires a responsible person and a written concept — the scenario moves out of §1.5 into §1.4.4 (first-party confirmation); if it acquires an identifiable stakeholder with acceptance authority — 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 without a second party in which one person signs the ACTZ for both sides	Violation of §5.5.3: the dual signature is not imitated; without a second party the ACTZ has no subject, the kind of §1.4.4 with confirmation: first-party applies.
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; the kind "first-party confirmation" §1.4.4 with 5 indicators, v1.1)	§1.4	§1.7.1
4	Negative scope: inapplicability contexts (4 contexts + 4 negative scenarios; §1.5.4 and scenario 2 redefined in v1.1)	§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 (12 types: ARCH/API/DATA/INT/PROC/UI/AI/SEC/OPS/TEST/DOC/UC)	§8.3	§4.4.1, §13.3.4

8	Normative TC principles (P1–P9)	§9.2	§13.3
9	TC types (<code>tc-type</code> : 6 values; <code>business</code> replaces the former <code>acceptance</code>)	§9.5	§4.5.2
10	Lifecycle states by object: description set (<code>draft</code> / <code>approved</code> / <code>superseded</code>), TR, ADAPT, TC implementation; BR / SR / SPEC / TC norm have no states of their own	§10.5– §10.9	§4.7–§4.10
11	Quality Gates: mandatory {QG-0, QG-1, QG-2}, optional {QG-3, QG-4} — transitions of the description set, TR, ADAPT and admission of a TC implementation	§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
17	Modal operators of the description language (5: MUST / MUST NOT / SHOULD / SHOULD NOT / MAY; introduced in v1.1 per §13.9)	§15.2	§11.5

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

Part of the RENAR Standard v1.1 · ← [Table of contents](#)

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 **normative**, but they are enforced differently: Claims 1 and 3 are anchored as the mandatory clauses [§13.3.1](#) and [§13.3.2](#); Claim 2 is a normative claim unfolded by the mandatory clauses [§13.3.1](#) and [§13.3.3](#) and has no enforcement point of its own.

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, Tessel, 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, Tessel, 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 `automation.set-version` field (the set version the implementation was written against) loses meaning without a stable version identifier (violation of V5).

- Acceptance of a BR against a set version (`accepted-outcomes[ ]` of the version record) 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.

Claims 1 and 3 are mandatory clauses (§13.3.1, §13.3.2); Claim 2 is unfolded through them and through §13.3.3 rather than checked as a separate item — the waterfall form is a property of how work is organized, not a binary fact about the substrate. 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

Part of the RENAR Standard v1.1 · ← Table of contents

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	the set draft → approved version N.M (§10.5.1)
V5 (cross-substrate version pin)	Pinning a description version from the implementation	the <code>automation.set-version</code> field in TC — the description set version (§9 , §10.5.4)
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 — the approved description set version ([chapter 10 §10.5](#)) or the approved state of a contractual artifact — 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:

- The `automation.set-version` and `last-run.set-version` fields in TC — the description set version (chapter 9 §9.3, §10.5.4).
- 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 (<code>_rev</code>)
V2 — atomic change unit	commit	commit	atomic revision	changelist submit	document update (single <code>_rev</code> 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	<code>_rev</code> reference + <code>created_by_order</code>
V6 — author + timestamp	commit metadata	commit metadata	revision properties	changelist metadata	doc fields (<code>author</code> , <code>updated_at</code>)

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 <code>mtime</code> 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)"	" <code>.src</code> pins the <code>.req</code> 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).

```
# 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 12 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	<code>automation.set-version</code> / <code>last-run.set-version</code> — a direct application of V5 to the set version
10 Lifecycle and QG	Status transitions — V2 + V3 + V4
11 Maturity model	RENAR-3+: V5 required; RENAR-4+: V6 + ai-provenance
13 Conformance	<code>RENAR-CONFORMANCE.yaml</code> — a mandatory artifact
guide/03	Practice on git
guide/04	Practice on a document store

04. Terms and Definitions

Part of the RENAR Standard v1.1 · ← 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 12 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
SPEC-UC	Use case: an end-to-end path through several SR / SPEC with an executor role (human / agent) and a reference of every step to a statement

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 <code>acceptance</code> 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 and the description set

BR and SR have **no** status of their own (§10.7): their state is derived from the set version — draft / approved / removed. The closed list of states of the **description set** (§10.5): `draft` → `approved` →

superseded ; the **major** flag distinguishes the presented version. "Verified" and "accepted" are properties of the pair (set version, product version), recorded in the version record (§10.5.4).

4.6.2 TR

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

4.6.3 SPEC

No status of its own — as for BR / SR (§10.7); structural completeness is a precondition of set version approval (§10.7.2).

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

A TC norm has no status of its own (§10.7). States of the TC **implementation** (§10.9): not admitted → admitted (QG-1) → **pass** / **fail** ; the implementation becomes stale when the norm is changed in a version later than **automation.set-version** . **pass** ↔ **fail** 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 ACTZ

A closed list (§7.13): **draft** → **sent** → **signed** ; **superseded** — when a previously signed protocol is revoked.

4.6.8 AR

A closed list (§7.4.6): **draft** → **issued** ; **superseded** — when the record is disavowed. The conditions for referencing it from **source.adversarial-review-ref** are stated there as well.

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

Any status outside the closed list for the corresponding object — the set, TR, ADAPT, TC implementation, ACTZ, AR — 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 a description set version, a task (TR), an ADAPT	Required
QG-1 — Implementation (§10.3.2)	Admission of a TC implementation to runs (TC implementation only)	Required
QG-2 — Verification (§10.3.3)	Verification of a set version on a product version; task → done	Required
QG-3 — Architecture (§10.4.1)	Approval of ADAPT (the Architect's signature) / architectural part of a set version	Optional (declared or absent)
QG-4 — Acceptance (§10.4.2)	Acceptance of the business outcome of a BR of a set version; task acceptance	Optional

Runner-managed transitions of a TC implementation (`admitted` → `pass` / `fail` and back) 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 — the approved set version (§3.3.3) — 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).

4.8.6 Description set and set version

Description set — the complete description of the system (BR, SR, SPEC, the normative parts of TC), approved and versioned as a single whole (chapter 10 §10.5). **Set version N.M** — an immutable version record that resolves into the content of all artifacts (§10.5.4); a minor version is released on every approved change, a major one is the presented version after a full audit. Individual set artifacts have no version or status of their own (§10.7); every reference to a version of the description — the `set-version` of a TC implementation, of a task, of the conformance manifest — points to the set version, and V5 pins the version of the record itself.

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
<code>ai-provenance.generated-by</code>	mandatory	Model identifier (<code><vendor>-<model>-<version>@<date></code>)
<code>ai-provenance.generated-at</code>	mandatory	UTC timestamp of generation (ISO-8601)
<code>ai-provenance.prompt-template</code>	mandatory	Substrate-native pointer to a prompt template (<code><path>@<version></code>)
<code>ai-provenance.context-tokens</code>	mandatory	Input context size (integer)
<code>ai-provenance.output-tokens</code>	mandatory	Output size (integer); source for the metrics in §12.3
<code>ai-provenance.human-edits</code>	mandatory	Boolean — whether manual edits were made after generation (informational field; see §4.10.1.1)
<code>ai-provenance.generation-time-ms</code>	optional	Generation latency in milliseconds; RECOMMENDED at RENAR-5 for cost/latency budget monitoring (§11.8.1)
<code>ai-provenance.cost-budget</code>	optional at RENAR-4, mandatory at RENAR-5	Planned generation cost budget
<code>ai-provenance.cost-actual</code>	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 object is outside the closed list of states of its kind (§4.6: the set, TR, ADAPT, TC implementation) 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 approved description set version it is conducted against (set-version , V5, §10.5.4); the base of comparison is the set version as a whole, not an individual SR / SPEC	Reconciliation hook RENAR-4+ (the machine channel) and the manual walkthrough record (§9.20) — the second channel, which sees undescribed behaviour; registered as a backward finding in a delta-ADAPT or as a candidate of §6.13

4.11.4	Implementation drift	The implementation substrate has stopped referencing the current version of the requirements substrate (the V5 pin is stale)	The TC implementation becomes stale (§10.9.4); no verification record of the version is created
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 norm has lost its verifies[] reference to an artifact of its set version, or the implementation's last-run.set-version does not match the version being verified	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[]	Subsystem BR	Cross-level edge to the parent system BR (§6.5.2, §6.8.2); not a parent edge
implemented-by[]	System BR	Auto-derived reverse edge for implements[] (§6.5.2)
implements-spec[]	TR	Implementation of SPEC-* specifications (§8.6.2)
constrained-by[]	SR	Constraints from SPEC-* (§8.6.1)
referenced-by[]	SPEC	Auto-derived reverse edge for constrained-by[] (§8.6.1)
depends-on[]	SPEC	Dependency graph between SPECs (§8.6.3)
supersedes	ADAPT / AR	A reference to the superseded artifact (§7.6.4)
superseded-by	ADAPT / AR	Auto-derived reverse edge for supersedes (§7.6.4)
verifies[]	TC	Closed list of verifiable artifacts with version (§9.3)

<code>verified-by[]</code>	BR / SR / SPEC	Auto-derived reverse edge from <code>verifies[]</code>
<code>source.adapt</code>	BR / SR / SPEC	The ADAPT from which the artifact was derived
<code>replaces / replaced-by</code>	Any	Replacement on deprecation (§10.5.3)
<code>supersedes</code>	A new version of an artifact	Which artifact is being replaced (in lieu of "reviving" an obsolete one)
<code>last-run</code>	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)	TK	Test Case (verifiable item)	Verification artefact	Story acceptance test
SPEC-* (12 types)	(RENAR-extension)	Design Description (subset)	Solution Component (subset)	Enabler tech spec (subset)

4.13.2 Lifecycle statuses

RENAR canonical	ISO/IEC 29148	CMMI
set draft (draft)	proposed	identified
approved set version (approved N.M)	agreed-to / baselined	committed

verification record of the version (§10.5.4)	verified	validated
BR acceptance in the version record (accepted-outcomes[])	accepted	accepted
removed from the version / superseded	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

Part of the RENAR Standard v1.1 · ← Table of contents

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** as an imitation of two parties; a project without a second party moves into the first-party confirmation kind, §1.4.4, where the ACTZ has no subject — §5.5.5).
- 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	Joint by SPEC type: Architect — overall; domain TL — expertise	QG-0: Architect or authorized role-holder

		lead		
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 the implementation is admitted to runs (QG-1, §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.

5.5.5 Combining roles

One person **MAY** combine roles — up to all the roles of chapter 5 in an organization without a second party (§1.4.4). The standard neither forbids nor compensates this: when roles are combined, adversarial checking by people is absent, and this is an accepted trade-off that the conformance claim **MUST** state (`confirmation: first-party` , §13.4.2). As separate people appear, they take their roles, and adversarial checking returns by itself.

Two restrictions are not lifted by combining:

1. **The ACTZ dual signature is not imitated** by one person (§5.5.1, §5.5.3). Without a second party the ACTZ has no subject — it is not signed unilaterally; decisions on the concept are recorded by a revised concept signed by the responsible person.
 2. **Execution and acceptance of one task are not combined:** the author of an implementation does not accept their own task and does not write the TC that checks it (P8, §9.18; AT authorship isolation, §9.19.2). This is a restriction on the task, not on the person: one person accepts others' tasks and executes their own.
-

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	The description set version (BR, SR, SPEC, TC norms — together), TR, ADAPT (on the Architect side)	Architect or authorized role-holder	Architect	AI critic (Reviewer)	Team
QG-1 Implementation Gate	The TC implementation (not admitted → admitted, §10.9)	Engineer / QA	Engineer	Automated runner	Team
QG-2 Verification Gate	The set version on a product version (the verification record, §10.5.4); TR	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)	Acceptance of a BR of the set version (accepted-outcomes[] of the version record, §10.4.2)	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 <code>architect-signature</code> field; <code>client-signature</code> and <code>vendor-signature</code> 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 v1.1 · ← 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.

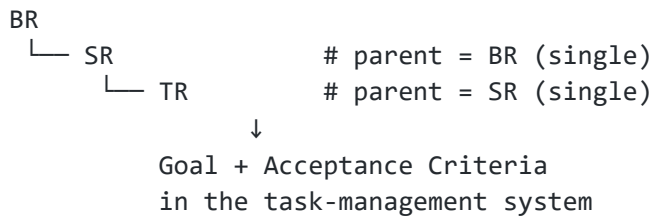
6.2 The closed list of three requirements-axis types

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



`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 removed from the next set version (§6.5.4, §10.5.3), 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
```

```

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

# === Ownership (mandatory) ===
# No status or version of its own: the artifact is approved as part of a
description set
# version, its state is derived from it ([§10.7](10-lifecycle-qg.md#10.7)).
owner: "<role / responsible person>"

# === Measurable business outcome (conditional) ===
# Mandatory if the implementation declared QG-4 ([§10.4.2](10-lifecycle-
qg.md#10.4.2)):
# that is the gate which checks the outcome was achieved. Without a declared QG-4
the
# field is omitted – acceptance of the business outcome is then not recorded
([§10.4.4](10-lifecycle-qg.md#10.4.4)). The full schema of the
# nested block – reference/02-schemas.md.
business-outcome:
  kpi-name: "<KPI>"
  measurement-method: "<how it is measured>"
  baseline-value: number
  target-value: number
  # the measured value and achievement are not here: a BR of an approved version
is immutable (§10.5.1);
  # QG-4 records them in accepted-outcomes[] of the version record (§10.5.4)

# === 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) ===
# Mandatory when level=subsystem AND the parent system has a BR in an approved
set version (§13.3.8).
# 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

# === Reusable component (conditional, §6.14) ===
assumptions: []                        # root BR of a component, path C: the list
of A-NN closed for the version (§6.14.2)
uses: []                               # root BR of a consumer: connected
components (§6.14.4)

# === 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>-<version>@<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>"                   # removal of the former artifact and the
back-reference replaced-by
                                     # are recorded in the set version record
(§10.5.3, §10.5.4)
---
```

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; the business-rule form — §15.6.2.

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 state — derived from the set version

A BR has no `status` field. Its state is derived from membership in description set versions ([chapter 10 §10.7](#)):

State	Condition	What is permitted
draft	Present only in the set draft	Editing by the author
approved	Belongs to the current approved version <code>N.M</code> — after QG-0 of the set (§10.5.2)	Task planning (TR) against the version; decomposition into SR is permitted already in the draft — a BR and its SRs are approved as one version (§10.5.2)
removed	Absent from the current version; the version record carries <code>replaced-by</code> if there is a replacement (§10.5.3)	Reading history only

"Verified" and "accepted" are properties not of the BR but of the pair (set version, product version): verification is recorded by an entry in the version record (QG-2, §10.3.3), acceptance of the business outcome — by `accepted-outcomes[]` (QG-4, §10.4.2, only when QG-4 is declared).

A removed BR is **not deleted** — it remains in the substrate history (V1) as a 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)

```
---
```

```
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

# === Ownership (mandatory) ===
# No status or version of its own: the artifact is approved as part of a
# description set
# version, its state is derived from it ([§10.7](10-lifecycle-qg.md#10.7)).
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.
quality-characteristic: functional-suitability | performance-efficiency |
compatibility |
                                interaction-capability | reliability | security |
maintainability |
                                flexibility | safety

# === Addressee of the obligation (conditional, §6.14.3) ===
applies-to: self                      # self | consumer; only in a component's SR;
default self

# === 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>-<version>@<date>"
  prompt-template: "<template-path>@<version>"
  context-tokens: integer
  output-tokens: integer
  human-edits: boolean

# === Replacement (mandatory if applicable) ===
replaces: "<old-id>"                # removal of the former artifact and the
back-reference replaced-by          # are recorded in the set version record

($10.5.3, $10.5.4)
---
```

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 <response>" (the operator "MUST" — §15.2). The precondition, trigger, or condition under which the requirement applies is expressed through a condition slot (§6.6.3.1).
Behavior	mandatory	A detailed description of observable behavior; functional scenarios.
Constraints	mandatory if applicable	Non-functional constraints (performance, security); full constraints live in the <code>constrained-by[]</code> SPEC.
Link to SPEC	mandatory if <code>constrained-by[]</code> is present	A short explanation of which aspects of behavior are governed by which SPEC.

6.6.3.1 Controlled form of the requirement statement (the condition slot)

The shell of a requirement is regulated in detail — frontmatter, graph edges, provenance, gates; the statement itself is not, although the agent deriving the implementation reads the **statement**, not the shell. Free prose admits four defects that no gate sees: a missing precondition, an unexpressed trigger, an undeclared unwanted behavior, and several statements glued into one sentence.

An SR statement SHOULD take one of six forms. The modality is advisory: the rule is introduced on v1.1 for review against practice, since it has no runs against accumulated description yet (§12.4 — the same argument about a rule without data).

Case	Form
Ubiquitous requirement	The system MUST <response>
State-driven requirement	WHILE <state>, the system MUST <response>

Event-driven requirement	WHEN <trigger>, the system MUST <response>
Unwanted behavior	IF <condition>, THEN the system MUST <response>
Optional feature	WHERE <feature is present>, the system MUST <response>
Complex requirement	WHILE <state>, WHEN <trigger>, the system MUST <response>

The condition marker stands as the first word of the sentence. It is recognized by **position**, not by typography — the rule is identical in both editions; the EN edition additionally renders the markers in uppercase per its own convention (§0.7), whereas the RU edition writes them in ordinary orthography because an uppercase convention in RU prose is prohibited.

Three particulars of the form matter:

- **WHERE means presence, not place.** The marker selects configurations in which the feature is included; it is not a question about location. The RU edition renders it as «При наличии» for the same reason — a literal «Где» would read as a question about place.
- **THEN is mandatory.** Without it, "IF ..., the system MUST" is grammatical but machine-indistinguishable from a conditional clause in free prose.
- **One statement — one sentence.** The form does not forbid several sentences in the section, but each carries exactly one statement: a glued statement is verified indivisibly, and a single TC then proves half of it.

The rule applies to SRs and to the statements of the type-specific sections of SPECs about the behaviour of the system (§15.6.1); operators, glossary discipline, prohibited constructions and atomicity — [chapter 15](#). A BR has its own form — "who, what, why" (§6.5.3), and for the "Constraints" section — the business-rule forms of §15.6.2; a TR has Goal and acceptance criteria (§6.7.3); the Pass/Fail criteria of a TC are governed by [chapter 9](#).

The basis is the EARS templates (Mavin et al., RE'09), named as a source in [reference/11](#) .

6.6.4 SR state — derived from the set version

Identical to BR (§6.5.4): no `status` field; draft / approved / removed are derived from membership in set versions (§10.7). Approving an SR as part of a version requires its `parent` BR to belong to the same version (§10.5.2). Verification is a property of the pair (set version, product version): all `verified-by` TCs passed with `last-run.set-version` equal to the version being verified (QG-2, §10.3.3).

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
  set-version: "N.M"                     # the set version the task is planned
against (§10.5.4; V5, chapter 3)

# === Link graph ===
implements-spec:                          # typed edges to SPEC
  - SPEC-API-NN
  - SPEC-UI-NN
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>-<version>@<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 <code>implements-spec[]</code> and to sections of the parent SR.

6.7.4 TR statuses

Status	Meaning	Trigger
<code>draft</code>	TR created, AC not yet finalized	Authoring
<code>approved</code>	AC approved, work may start	QG-0 (chapter 10): goal + AC present
<code>done</code>	AC verified, TCs passed on the task's set version	QG-2; <code>verified-by TC pass with last-run.set-version = source.set-version</code>
<code>obsolete</code>	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
Mandatory	<code>BR.level = subsystem</code> AND <code>scope.system</code> has ≥ 1 BR in an approved version of its set	<code>implements[]</code> MUST contain ≥1 reference to an applicable BR of the parent system
Permitted	<code>BR.level = subsystem</code> AND the parent system is a container with no BR of its own (organizational-level scope)	<code>implements[]</code> is omitted; the rationale is recorded in the <code>Context</code> section with a reference to ADAPT§
Prohibited	<code>BR.level = system</code>	<code>implements[]</code> 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 belongs to the approved set version of its system (§10.11.1); a target present only in a draft is fatal.
- Detect cycles in **implements** chains; a cycle is fatal.
- On removing a target BR from a version (§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 removed from the next set version (§10.5.3) with the reason given in the version record (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.

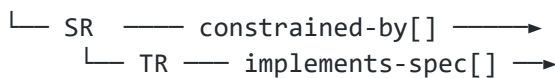
6.10.1 Link to ADAPT

`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):

BR



Parallel specifications axis:

```

SPEC-ARCH  SPEC-API  SPEC-DATA
SPEC-INT    SPEC-PROC SPEC-UI
SPEC-AI     SPEC-SEC  SPEC-OPS
SPEC-TEST   SPEC-DOC
  
```

Edge	Type	Obligation
<code>SR.constrained-by[] → SPEC-*</code>	Graph (multiple)	Optional; present when governing SPEC exist
<code>TR.implements-spec[] → SPEC-*</code>	Graph (multiple)	Optional; specifies SPEC for the implementer
<code>SPEC-*.referenced-by[] → SR / TR</code>	Auto-derived inverse	Auto-computed by substrate-native indexing

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
      |               |
      |               |— 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
      |
      |— §6.8.2)

```

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
      |
      |— §6.8.2)

```

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
```

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)	Set version (draft → N.M) for BR / SR; TR (draft → approved)	frontmatter valid, mandatory fields and body filled, identifier unique (V1); <code>source.adapt</code> , if present, points to an approved ADAPT, otherwise <code>source.adversarial-review-ref</code> is present (BR / SR); <code>parent</code> belongs to the same version (SR); for every assertion describing observable behaviour — a pair of TC norms (§9.7); adversarial review performed; the Architect's signature	BR / SR belong to the approved version N.M ; decomposition and task planning are permitted; TR — approved
QG-2 (Verification)	Set version × product version (BR / SR); TR (approved → done)	All TCs of the version pass with last-run.set-version = N.M ; red history; mandatory TC kinds per SPEC type	Verification record in the version record (§10.5.4); TR — done

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 verification of a set version and the TR transition into `done` are 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: the state of a BR / SR is derived from the set version ([§6.5.4](#)), the TR state machine is `draft` → `approved` → `done` | `obsolete` (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**. The second regular source of candidates of this class is a manual walkthrough finding (§9.20): behaviour seen by a person walking a scenario and not described in the set, when per §6.13.2 it is not observable by the client.

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 Reusable component

6.14.1 A component is a system in its own tree

An artifact created not for one client but for repeated use (a library, a platform, an internal engine) is a **system in its own tree** (§6.8.2 — the same precedent as a subsystem that is an independent product), not a fourth level of scope. A component has its own BR (§6.4: no business goal — no project), its own SR / SPEC / TC norms and its own description set (§10.5). The business side of a component is represented by one of two paths — both permitted, the choice determined by the structure of the organization, not by the size of the team (§5.5.5):

The path labels come from ADR-016, where path A (a fourth level of scope) was rejected.

Path	The client side of the component	Conformance of the component
B — the component's Product Owner	appointed inside the organization; signatures per the general rules of chapter 5	self-standing
C — an assumed side	the component declares assumptions about the consumer (§6.14.2); every consumer confirms them on connection (§6.14.4)	conditional : completed by the connection — the component MUST NOT claim conformance on its own, the consumer project claims it

The external prototype of path C is an element developed out of the context of a specific system (ISO 26262 SEooC): its requirements derive from declared assumptions about the future system, which the integrator confirms.

6.14.2 Declared assumptions

The root BR of a component (`level: system`) carries a list `assumptions[]` , closed for the given set version — assumptions about the consumer on which its statements rest: what the consumer provides itself (for example, centralized secret storage), the environment the component runs in, the obligations the consumer takes on. Every assumption has an immutable identifier `A-NN` (V1), a wording and a statement of how the consumer confirms it. An assumption is not a backward finding: it has no client addressee before connection; it is **contestable** in the adversarial review (§7.10.2) — the reviewer checks whether something that ought to be a requirement is hidden among the assumptions.

6.14.3 The `applies-to` property

A statement of a component's SR or SPEC carries the property `applies-to: self | consumer` (default `self`):

- `self` — an obligation of the component itself; proven by its TCs once, the proof is inherited by consumers.
- `consumer` — an obligation of the **consumer** (how it **MUST** use the component); at every consumer it is covered by a static TC (`automation.kind: static` , §9.8.1) — no new TC

kind is introduced. The red history (§9.18.2) for such a TC is achieved by a fixing run on a deliberately violating configuration: a TC born green is not evidence.

consumer statements are a scarce resource: each is multiplied by the number of consumers and their dependencies (the transitive closure over the **uses** graph), whereas a **self** statement is proven once. Only the proof of the component's **self** statements is inherited; the behaviour of the application **with** the component is not inherited and is proven by the consumer's TCs.

6.14.4 The **uses** edge

A consumer connects a component through a typed **uses[]** edge in its root BR (**level: system**) — **not a parent edge**; the prohibition of multiple parents (§6.8.3) does not apply to it:

```
uses:
- component: "<system-id of the component>"
  set-version: "N.M"                # the component's set version; mandatory
(V5)
  assumptions-confirmed:             # every A-NN of the component – by name
    - { id: A-01, confirmed-by: "<actor>", confirmed-at: "<ISO-8601>" }
```

Normative rule of **uses[]** :

1. **The version is mandatory**: the edge names an approved set version of the component (§10.5.4); without it a new release of the component would silently change the meaning of the consumer's requirements.
2. **Read only**: the component's tree is immutable for the consumer; a divergence is a backward finding in the consumer's contour or a request to the component's owner.
3. **Confirmation of assumptions**: every **A-NN** of the component version is confirmed by name by the side responsible for the consumer's description (the stakeholder or the responsible person, §1.4.4); an unconfirmed assumption — the consumer's set version is not approved.
4. **Acyclicity** of the **uses** graph; a cycle is fatal.
5. **Locality of constrained-by[]** : the consumer's references to the component's artifacts are expressed only through **uses** ; targets in a foreign tree in **constrained-by[]** are prohibited.
6. **Transitivity**: the consumer confirms the assumptions of its direct components only — those in its **uses[]** ; the assumptions of components connected through them are confirmed by their immediate consumer, and that confirmation is inherited together with the proof of the **self** statements. The **consumer** obligations of all components in the transitive closure of **uses** the consumer **MUST** cover with its own static TCs (§6.14.3).
7. **The stop rule**: having found an incompatibility of the **consumer** statements of two components connected by one consumer, the consumer **does not approve the set version** until a person has decided; the agent does not resolve such an incompatibility itself. The decision is recorded in the consumer's description — by changing the connection (another version, or removing the component from **uses[]**) or, if client-observable behaviour is affected, by a backward finding with a contractual outcome (§6.13.2). The substrate does not compute this rule: the incompatibility of statements is established by a person in the adversarial review (§7.10.2).

8. **Changing the component version:** a new component version is, for the consumer, a change of its own set (`set-version` in the edge) with re-confirmation of the changed assumptions; the removal of a component artifact gives the consumer a warning, not a cascading removal (after the pattern of `implements[]` , §6.8.2).

6.14.5 What is not introduced

A fourth level of scope, a "product-originated" provenance class, separate "library / framework" types and a new TC kind are not introduced: the distinction is expressed by the `applies-to` property, the set version and the `uses` edge. No closed list of §1.7.5 is extended; the mandatory clauses of §13.3 are not affected.

6.15 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
05 Roles	The client side of a reusable component (§6.14.1): the component's Product Owner or declared assumptions confirmed by the consumer; combining roles — §5.5.5
10 Lifecycle and QG	The description-set cycle (§10.5): the BR / SR state is derived from the set version; QG-0 / QG-2 for the requirements axis (QG-1 — for TC only); the implements and uses edge enforcement points (§10.11.1)
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

Part of the RENAR Standard v1.1 · ← Table of contents

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

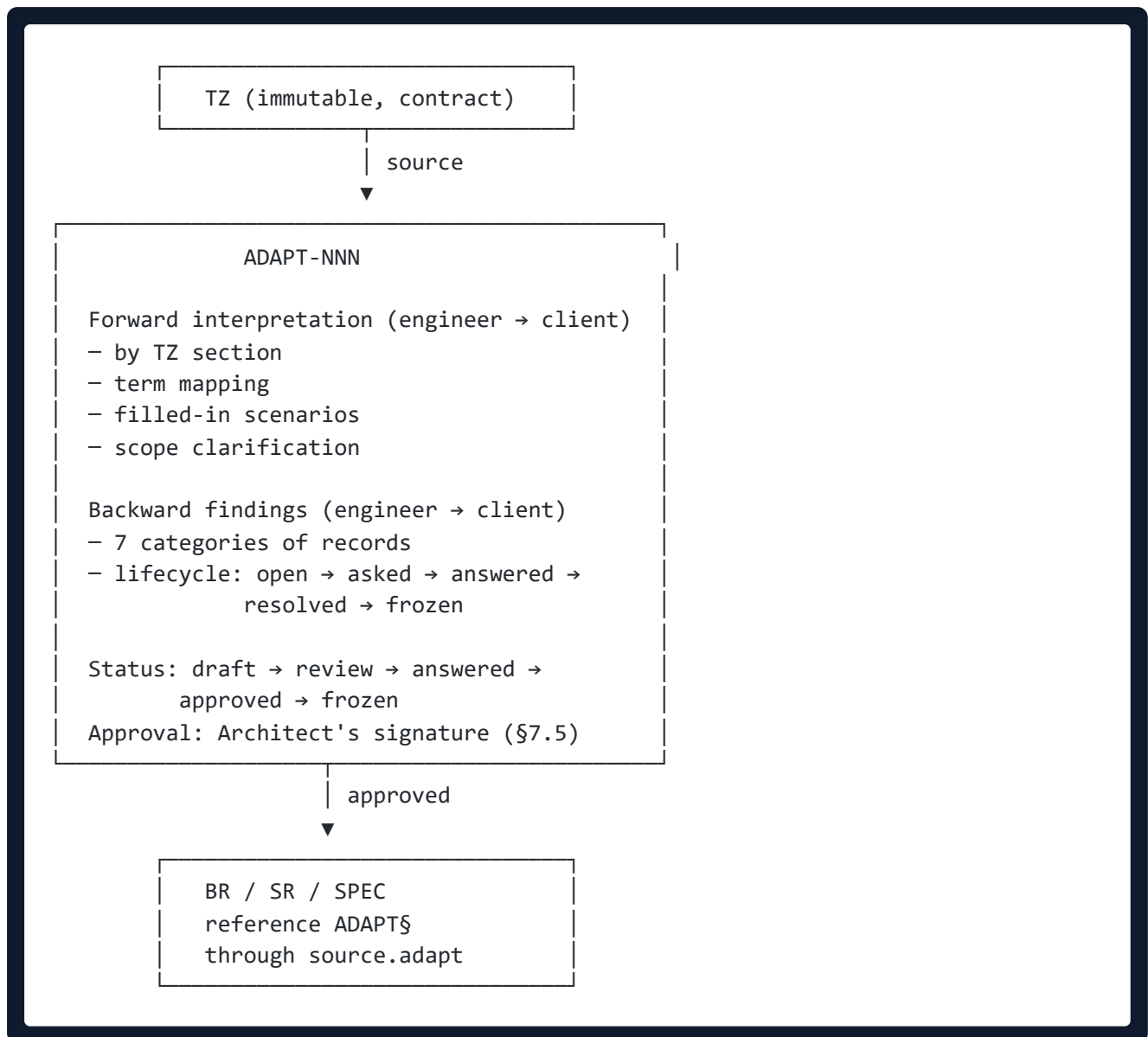
Without a formalized intermediate artifact between the TZ and BR/SR, one of two negative outcomes arises:

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)
--	--------------	--------------------

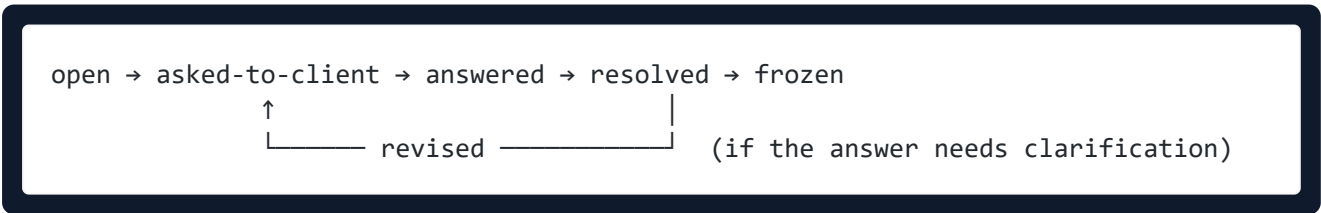
7.4.4 Backward findings and their categories

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



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). In the first-party confirmation kind (§1.4.4) the ACTZ has no subject: `decided-in` points to the section of the revised concept signed by the responsible person (V6) — with the same obligation and the same prohibition on referencing unsigned text. 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
<code>id</code>	REQUIRED	<code>AR-NNN</code> ; immutable

tz-ref	REQUIRED	The (delta-)TZ under review
trigger-stage	REQUIRED	Review stage: <code>import-tz</code> / <code>decompose-br</code> / <code>decompose-sr</code> / <code>spec</code> / <code>tc</code> (the list is aligned with ADAPT's <code>trigger-stage</code> , §7.8.1)
<code>reviewer.vendor</code> + <code>reviewer.model</code>	REQUIRED	The adversarial reviewer's model; it MUST differ from the primary agent's model (§7.10.2)
<code>primary.vendor</code> + <code>primary.model</code>	REQUIRED	The primary agent's model at the moment the AR was issued — the second operand of the §7.10.2 condition. Without it, independence is checkable only at issuance: later there is nothing to compare against, and when the primary model changes the review ceases to be adversarial without any signal. Format as for <code>ai-provenance.generated-by</code> (§4.10.1), including the version
verdict	REQUIRED	<code>findings-present</code> or <code>no-findings</code>
<code>produces-adapt[]</code>	Conditional	REQUIRED and non-empty when <code>verdict: findings-present</code> ; absent or empty when <code>no-findings</code>
status	REQUIRED	<code>draft</code> / <code>issued</code> / <code>superseded</code>
<code>superseded-by</code>	Conditional	REQUIRED when <code>status: superseded</code>
<code>signature.author</code> + <code>signature.timestamp</code>	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
<code>draft</code>	The review is conducted, the verdict is being drawn up; referencing this AR from <code>source.adversarial-review-ref</code> is prohibited
<code>issued</code>	The verdict is issued and signed by the reviewer (V6); the record is immutable
<code>superseded</code>	A later review at the same stage displaced the earlier verdict; a terminal state, the record is retained for audit (V1)

An AR in status `issued` 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 <code>resolved</code> and carries <code>decided-in</code> 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 Preconditions of a delta-ADAPT

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. The standard does not set the order of work (§1.2.1); the preconditions are normative:

- **The delta-TZ is registered** in the substrate as a new immutable document (`TZ-YYYY-NNN-delta-N`) and **signed by the client** (V6 author + timestamp) — without this, artifacts referencing the delta are not approved.
- **The adversarial review of the delta-TZ is performed** (§7.10.2) and its verdict is recorded in the substrate with V6 author + timestamp — machine-accessible for audit (hook §10.11.1 reference-validation).
- **With the verdict "findings present"** a delta-ADAPT (`ADAPT-NNN-delta-N`) exists with the fields `parent-adapt: ADAPT-NNN` and `source-tz: TZ-YYYY-NNN-delta-N` : the forward interpretation covers only the sections of the delta-TZ, backward findings are recorded against the delta-TZ, approval is the architect's signature §7.5; the client's decisions on the findings — in new ACTZ with a dual signature. The affected BR / SR / SPEC reference the delta-ADAPT through `source.adapt` .
- **With the verdict "no findings, no clarifications"** no delta-ADAPT is created; the BR / SR / SPEC changed as a result of the delta-TZ reference `source.tz-section: TZ-YYYY-NNN-delta-N` directly.

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 **frozen** and from **obsolete**, the state through which TR / SPEC / TC become outdated (ADAPT has no **obsolete** state, §10.8.1). **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`

```
# 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)

```

---
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>-<version>@<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 <code>open</code>	All backward findings in <code>open</code> or <code>asked-to-client</code>
QG-ADAPT-asked	All backward findings in <code>asked-to-client</code> ; the questions have been put to the client as part of an ACTZ (<code>status: sent</code>)	Awaiting the protocol's signing
QG-ADAPT-answered	All backward findings in <code>answered</code> ; resolution begun	Ready for finalization

QG-ADAPT-approve	All backward findings in <code>resolved</code> and carry a <code>decided-in</code> pointing to a signed ACTZ; the Architect's signature is ready	ADAPT immutable; generation of BR / SR / SPEC is permitted
QG-ADAPT-frozen	<code>approved</code>	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 description set version <code>N.M+1</code> is approved (§10.5), 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 <code>source.tz</code> -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 — the next set version (§10.5) — 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.

The form of a clause. An ACTZ clause is an obligation to the client and is worded as an obligation, without interpretive hedges; a question requiring the client's decision is put as a question (§15.6.4).

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
<code>id</code>	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 $\geq 80\%$**). 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 <code>source.adapt</code>
08 Specifications	SPEC-* also reference ADAPT through <code>source.adapt</code>
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 — 12 SPEC types

Part of the RENAR Standard v1.1 · ← 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 twelve specification types, and the list is closed: `SPEC-ARCH` , `API` , `DATA` , `INT` , `PROC` , `UI` , `AI` , `SEC` , `OPS` , `TEST` , `DOC` , `UC` — a new type is introduced only through the formal change procedure of the standard ([chapter 13](#)).

A specification describes **four different subjects**, and they MUST NOT be conflated. Nine types describe **how the system is built**: `SPEC-UC` — **a path through it** ([§8.5.12](#)). `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	<code>SPEC-TEST</code>	On what bench and on what data conformance is proven (§8.3.2)
Delivery	<code>SPEC-DOC</code>	What enters the delivery to the client besides the system itself

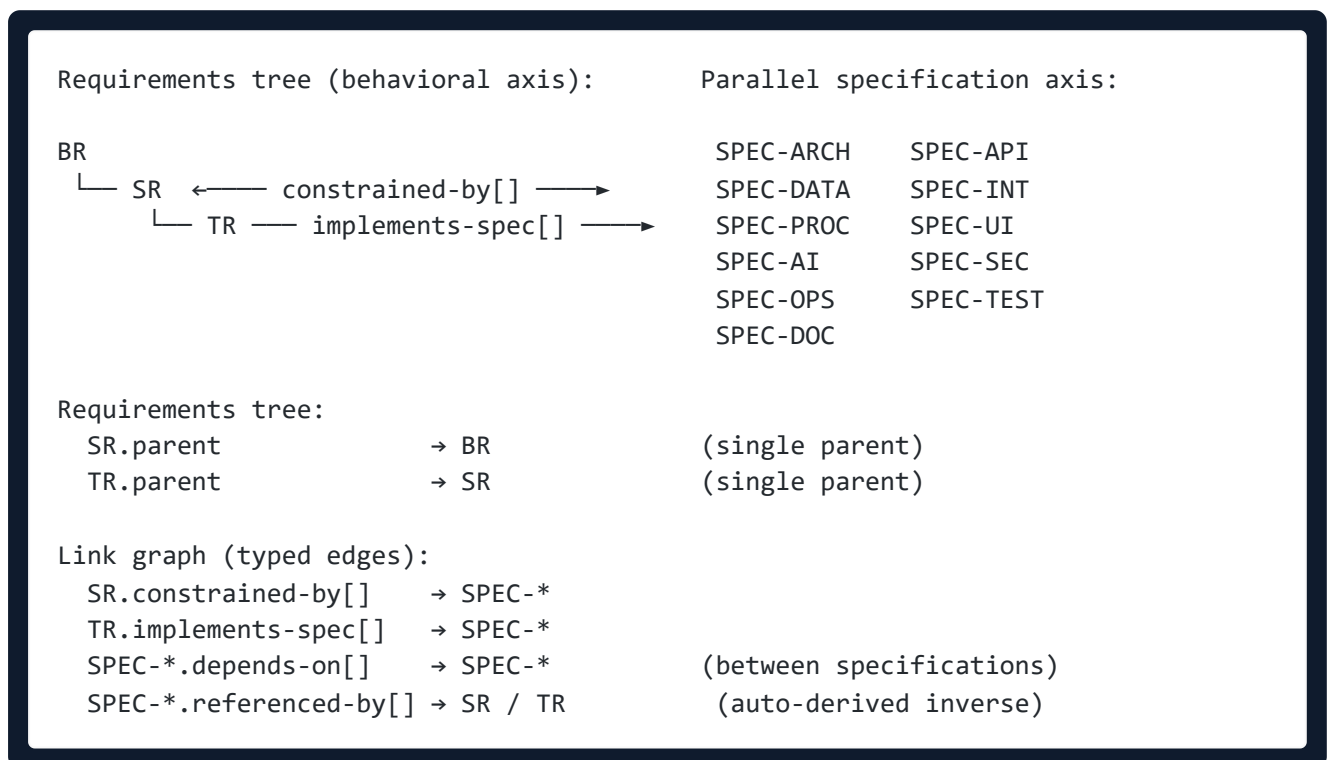
8.2.2 SPEC as a parallel axis: links through a typed graph

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).



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 twelve 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
SPEC-UC	Use case: an end-to-end path of a user or an agent through several SR / SPEC; the executor role (human — through the interface, agent — through the API) sets the run channel of the covering TCs; every step references the statement it touches	Use cases (Cockburn), ISO/IEC/IEEE 29148 §6.4 (scenarios), Gherkin (step form)

Mandatory types. A description set of the **system** level contains non-empty **SPEC-ARCH** and **SPEC-SEC** regardless of whether the TZ mentions them; their absence or an empty mandatory body is a violation of the structural completeness of the version (§10.7.2, §10.7.3). The silence of the TZ about security or about the structure of the system is a gap, not a client's decision: the description closes it with a statement of its own (with client-observable consequences — through a backward finding and the contractual contour, §6.13.2), not inherits it. The remaining types apply where there is a subject (chapter 11).

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 <code>compliance-refs[]</code> , 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. §10.9.4 marks a TC implementation stale on **any** change of the norm it references, automatically and without inspecting the cause. If a TC norm referenced the bench through SPEC-OPS, then an edit to the **deployment procedure** — unrelated to the bench — would make the implementations of **every** TC referencing that SPEC-OPS stale. 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 12 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

# === Ownership and priority ===
# No status or version of its own: a SPEC is approved as part of a description
set
# version, its state is derived from it ([§10.7](10-lifecycle-qg.md#10.7)).
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)

# === Addressee of the obligation (conditional, §6.14.3) ===
applies-to: self                  # self | consumer; only in a component's
SPEC; default self

# === 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>-<version>@<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>"              # removal and the back-reference replaced-by
– in the set version record (§10.5.3)
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.

Coverage completeness. The mandatory body MUST describe the subject of the specification exhaustively. Restricting coverage to a subset of elements by a subjective selection criterion is prohibited

— in all twelve types and regardless of the wording used. The prohibition applies to the subject, not to the vocabulary: an evaluative adjective ("key", "critical", "main") or a phrase of measure ("to a sufficient extent", "essential", "relevant") is prohibited only where it narrows what is subject to description. The same words remain legitimate where they name a property of the subject itself — such as the severity class of alerts in §8.5.9.

The subject of a specification is checkable: it is everything named by the linked requirements (the "Link to requirements" section) and, for a SPEC-UI, by the screen list (§8.5.6.1); narrowing is permitted only by an explicit Scope section listing the excluded elements and the reason for each exclusion.

The rationale is the inversion of the source of truth (§2.3.1): whatever did not enter the SPEC as "insignificant" has no source of truth at all. An agent implementing the system is left with two ways out, and both violate the standard: leave the undescribed part unimplemented (product incompleteness) or implement it arbitrarily (guesswork, prohibited by §2.3.3).

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 every L2 container, quality attributes (latency / throughput / availability), ADR log, the list of the system's screens — the code, name and source of every screen from the description (§8.5.6.1); a system without an interface records that explicitly.

Spec-specific TC ([chapter 9](#)): architecture conformance tests (zoning), reference tests of quality attributes. Coverage of the screen list is not a TC but a precondition of version approval (§10.7.2, §8.5.6.1).

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 (every scenario step with a reference to the statement it touches, §8.5.12.1), 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, `screens[]` (the codes of the screens from the SPEC-ARCH list that the document covers, §8.5.6.1).

Mandatory body: overall interface structure, all screens in the declared Scope — each under a code from the SPEC-ARCH screen list (§8.5.6.1), user journeys without technical details covering every user action available through the interface (every journey step with a reference to the statement it touches, §8.5.12.1), 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).

8.5.6.1 Screen and completeness across the system

Coverage completeness (§8.4.1) acts inside the declared Scope of one document and does not guarantee that the sum of all SPEC-UIs covers the interface of the system: for a document that does not exist there

are no norms it could violate. Completeness across the system is given by the screen list and the coverage rule.

A screen is a state of the interface in which the user performs an action or observes its result, addressable independently of the implementation technique: a route, a page, a modal window, a wizard step, a screen of a mobile or desktop application — when an action is performed or its result is shown in it. A state without an address of its own — a waiting indicator, a pop-up notification over a screen — is not a screen: it is described by the screen on which it appears.

The screen list is kept in the mandatory body of SPEC-ARCH (§8.5.1): an immutable screen code (V1), a name, a source — the SR or the section of the SPEC-ARCH itself from which the screen follows. The source of the list is the description only. A list built from the routes of the implementation would make the code the authority on "what MUST be described" and would violate the inversion of the source of truth (§2.3.1, the prohibition of §2.3.3); a screen that exists in the implementation and is absent from the list is a source-of-truth drift (§4.11, class 4.11.3) and a backward finding, not grounds to extend the list after the fact. A system without an interface records "no interface" in the list — explicitly, not by omitting the section.

The coverage rule. Every screen of the list is covered by at least one SPEC-UI of the same set version: a SPEC-UI declares the codes of the screens it covers in `screens[]` and describes each in its "Screens" section under the same code. A screen of the list without a covering SPEC-UI is a violation of the structural completeness of the version (§10.7.2): the version is not approved until the screen is described or removed from the list by a decision visible in the version record. A code in `screens[]` absent from the list is the same error from the other side. The number of SPEC-UIs is not limited; completeness inside each is per §8.4.1. The rule was introduced in v1.1 (ADR-023 rev. 3); it introduces no new artifacts or types.

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.

8.5.12 SPEC-UC

Type-specific frontmatter: `role` (`human` — the path is walked by a person through the interface; `agent` — the path is walked by an agent through the API / CLI; the field sets the run channel of the covering TCs), `persona` (a reference to the persona section of the ADAPT, for `human`), `covers[]` (the SR / SPEC the scenario stitches together; at least two), `entry` (the SPEC-UI screen or SPEC-API operation the path starts from), `steps[]` (one entry per step: `n` , `action` , `ref` — the address `<id>#<n>` of the SR / SPEC statement the step touches (§15.1.1), `mandatory`; `expects` — the observable result of the step).

Mandatory body: the goal of the scenario and its executor (role, persona); preconditions (state of the system and data); the main path — numbered steps, each with a reference to the statement it touches (§8.5.12.1); alternative and negative branches (what happens on failure at each step, with the same reference); postconditions; covering TCs (which TCs walk the path step by step and end to end).

Spec-specific TC: for `role: human` — a `ux` TC per step and an end-to-end journey E2E; for `role: agent` — `system` / `contract` per the channel of the step (§9.8). Negative branches — separate TCs, mandatory on a par with the steps of the main path (§9.7).

Why a separate type. A user journey in SPEC-UI and a happy path in SPEC-PROC live inside a document of their own type and do not leave its Scope; a use case by definition stitches several SR and SPEC together and carries a property of its own — the executor role, on which the run channel depends: a person's path through the interface is checked by `ux` TCs, an agent's path through the API — by `system` / `contract` . Bypassing the interface with a `system` TC for a green run while the interface is broken is the observed failure for which the type and the rule of §9.8 were introduced. The type was added in v1.1 under the procedure of §13.9.

8.5.12.1 Step reference to a statement

Every scenario step — in SPEC-UC, in a SPEC-UI user journey (§8.5.6) and in the happy path / alternative scenarios of SPEC-PROC (§8.5.5) — MUST carry a `ref` to the SR or SPEC statement it touches, in the

address form `<id>#<n>` (§15.1.1); `#<n>` is omitted only for an artifact with a single statement. In a SPEC-UC the reference is the `steps[].ref` field; in a SPEC-UI user journey and SPEC-PROC scenarios the steps are written as a numbered list, and every step ends with the reference `→ <id>#<n>` — that is how the structural-completeness check finds a step without a reference in prose. A step without a reference violates structural completeness (§10.7.2); a purely navigational step references the SPEC-UI statement about the screen or transition. The reference gives the reverse answer — which statements are touched by no scenario — and makes the TC coverage of a step checkable: the TC covering a step names the same address in `verifies[]` (`id` + `statement` , §9.3).

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 the change is compatible and the downstream enters the new version unchanged, or the downstream is edited in the same draft; the TC implementations affected by the change become stale (§10.9.4), and no verification record of the new version is created until they are run.

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 <code>specs/ui/</code>
AIC-NN	SPEC-AI-NN	Rename ID + move to <code>specs/ai/</code>
INT-SR-NN	SPEC-INT-NN	Rename ID + move to <code>specs/int/</code>
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 12 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 no state machine of its own: its state is derived from the description set version (chapter 10 §10.7). Conditions by stage:

Stage	Condition
draft	Created; mandatory frontmatter fields are being filled; present only in the set draft
structural completeness	Mandatory body sections (§8.4.1) and type-specific ones (§8.5) are present — a precondition of version approval (§10.7.2)
approved	Belongs to the approved version <code>N.M: depends-on[]</code> is consistent within the same version (§10.7.3); the Architect's signature on the version
verified on a product version	All mandatory spec-specific TCs (chapter 9 §9.8) passed with <code>last-run.set-version = N.M</code> — QG-2 of the version (§10.3.3)
removed	Absent from the current version; <code>replaced-by</code> is mandatory in the version record (§10.5.3)

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 12 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	<code>SR.constrained-by[]</code> , <code>TR.implements-spec[]</code>
07 ADAPT	SPEC references ADAPT through <code>source.adapt</code>
09 Test cases	Spec-specific TC types (the table of mandatory TC kinds for each SPEC type)
10 Lifecycle and QG	The SPEC state is derived from the set version (§10.7); preconditions by SPEC type at version approval (§10.7.3)
03 Substrate versioning	SPEC IDs are immutable (V1); migration is atomic (V2)
11 Maturity model	RENAR-3+: all 12 SPEC types where applicable
reference/02 — schemas	Full machine-readable schema for each type-specific extension
reference/05 — knowledge graph schema	<code>constrained-by[]</code> , <code>implements-spec[]</code> , <code>depends-on[]</code> as edge types in the graph

09. Test Cases

Part of the RENAR Standard v1.1 · ← 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 --> I
  G --> I
  H --> 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 <code>tests/</code> 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 <code>automation.location</code> 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 admitted TC implementations (QG-1) are automated. The run is performed by an automated runner (CI / AI-runner / specialized executor). Results in <code>last-run</code> 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	<code>last-run</code> — bot-managed	The <code>last-run</code> field (date / result / runner-id / set-version / judge-report) is filled in only by the automated runner. Manual editing of <code>last-run</code> 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 TC norm is approved in a version and the implementation admitted (QG-1) 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 <code>[test-spec-change]</code> 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

# A TC norm has no status of its own: it is an item of the description set
# ([§10.7](10-lifecycle-qg.md#10.7)).
# Implementation and run states – automation.* and last-run.* ([§10.9](10-
# lifecycle-qg.md#10.9)).

# === Verification target (mandatory; at least one) ===
verifies:
  - id: SR-NN | BR-NN | SPEC-<TYPE>-NN    # the description version is stated once
  - automation.set-version
    statement: 5                          # the statement address <id>#n (§15.1.1);
mandatory when the artifact has more than one statement
  - 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
  set-version: "N.M"                                    # the set version the implementation was
written against (QG-1, §10.9.2; V5)
  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>"

```

```

set-version: "N.M" # the set version the run was performed on
judge-report: "<inline or pointer>"

# === AI provenance (mandatory at RENAR-4+; canonical schema – §4.10.1) ===
ai-provenance:
  generated-by: "<vendor>-<model>-<version>@<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 (mandatory if applicable) ===
replaces: "<old-id>" # removal of the former TC norm and
replaced-by – in the set version record (§10.5.3)
---
```

`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). The description version is stated once per implementation — `automation.set-version` : the set version the implementation was written against (V5, chapter 3 §3.3.5; §10.5.4). QG-2 of the version (§9.10) requires `last-run.set-version` equal to the set version being verified.

`automation.status` takes exactly two values. `manual-pending` is a temporary placeholder: the mandatory `manual-pending-until` / `manual-pending-reason` and the deadline ceiling (§10.9.2) apply to it alone; a permanently manual run as a third status is not provided for. How the evidence of a run — automated or manual — is recorded is not governed by the standard: specific test runners are excluded from the normative scope (§1.3, item 5); only the mandatory presence of the `automation.location` and `last-run.run-ref` pointers is governed (§9.12).

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 <code>tc-type: ux</code> — 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 is a violation of structural completeness: the set version is not approved (§10.7.2).

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.

The triple "Given — When — Then" is a reading of three of the seven sections (preconditions, steps, Pass criterion), not their replacement; the language of criteria and steps — chapter 15, the correspondence — §15.6.3.

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 $\geq N\%$ against the baseline).

Judge $\neq$ 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 $\geq 10\%$ 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
<code>negative: false</code>	<code>negative: true</code>
Describes the happy path / success behavior	Describes boundary conditions, violations, bypasses
<code>paired-with: [TC-<neg-id>]</code>	<code>paired-with: [TC-<pos-id>]</code>

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-0 of the set (§10.5.2) MUST block the approval of a version in which at least one normative statement is not covered at all or is covered only by a positive TC norm; the presence of coverage is checked by the

coverage-presence enforcement point (§10.11.1) — mandatory clause §13.3.5.

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 norm of each "mandatory kind" — presence is checked at the approval of the set version (QG-0, §10.5.2), passing — at the verification of the version (QG-2, 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 <code>contract-file</code>)	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 <code>tc-type: contract</code> (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 $\neq$ 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 (<code>automation.kind: static</code>): 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 / <code>eval</code>

SPEC-UC	role: human — a ux TC per step + end-to-end journey E2E; role: agent — system / contract per step by channel	Negative branches as separate TCs (mandatory, §9.7); SLA of the end-to-end path
---------	---	---

The enforcement point of set version approval (§10.11.1, **coverage-presence**) MUST check for the presence of at least one TC norm of each mandatory kind and block the approval in their absence; passing these TCs is the subject of QG-2 of the version (§10.3.3).

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; the SPEC-UC row was added in v1.1 under the procedure of §13.9. Extension is only through the formal change procedure of the standard (chapter 13).

Type of the covering TC for the interface (mandatory). Every SPEC-UI statement describing an action available through the interface (a screen, an element, a user-journey step) and every SPEC-UC step with **role: human** MUST be covered by at least one TC of type **ux** (§9.6.1). A **system** TC checking the same behaviour while bypassing the interface **does not** close such a statement: a green run that bypasses the interface while the interface is broken is the observed failure for which the rule was introduced. For a SPEC-UC step with **role: agent** the covering TC is **system** or **contract** by the channel of the step. The rule does not extend §13.3 (§13.3.5 governs the presence and pairing of coverage, not its type) and introduces no new TC kind.

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:

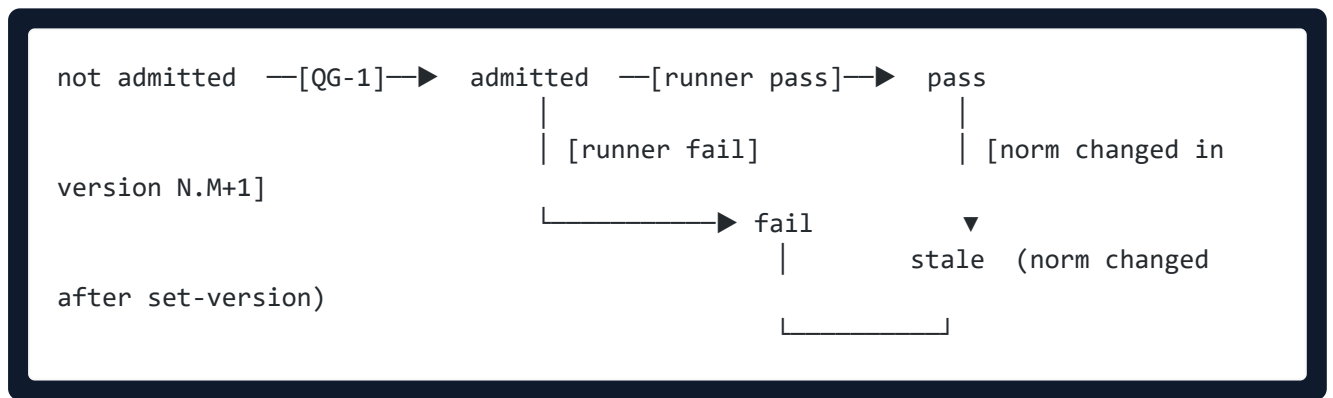
Cause of failure	What changes
The code violates a rule written in the SPEC	The code
The rule is judged wrong or in need of tightening	The SPEC — a new version (not an edit to the test!)
The analyzer yields a false result	The tool ; the baseline is untouched

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 Norm and implementation — two different objects

The normative part of a TC (frontmatter and body per §9.4) is an item of the description set: it has no status of its own, is approved as part of a set version and is removed from it per §10.5.3. A lifecycle of its own belongs to the **implementation** — the check code (§10.9):



Implementation state	Meaning	How it is recorded
not admitted	The implementation is absent or has not passed QG-1	<code>automation.status</code> not confirmed by a dry-run
admitted	The dry-run passed; the red history is recorded; <code>automation.set-version</code> points to the approved version of the norm	QG-1 (§9.10)
<code>pass</code> / <code>fail</code>	Result of the last run on <code>last-run.set-version</code>	Bot-managed upon the run (§9.12)
stale	The norm was changed in a later set version; a run against the previous norm is not evidence	The norm was changed in a version later than <code>automation.set-version</code> (§10.9.4)

A removed TC norm is preserved in the substrate history as a trace (V1): the substrate implementation **MUST** keep the TC identifier immutable and **MUST NOT** allow its reuse for a new TC norm (chapter 3

§3.3.1).

9.9.2 Relation to the verification of the set version

"Verified" is a property not of a BR / SR / SPEC but of the pair (set version, product version) (§10.3.3): all TC norms of version `N.M` have an admitted implementation with `last-run.result = pass` and `last-run.set-version = N.M`. The runner keeps the verification record in the set version record (§10.5.4).

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 norm — an item of the set version being approved	For every normative statement of the version — a pair of TC norms (§9.7); <code>verifies[]</code> points to artifacts of the same version; mandatory body sections (§9.4) filled; the approver's decision recorded substrate-natively (V3 + V6)	The TC norm belongs to version <code>N.M</code> ; implementation against it is permitted
QG-1 — verification implementation (§10.3.2)	TC implementation — admission to runs	<code>automation.set-version</code> = the approved set version containing the norm; <code>automation.status</code> = automated (or manual-pending with a deadline and reason); the runner dry-run passed; red history (§9.18.2)	The implementation is admitted; production runs are permitted
QG-2 — verification (§10.3.3)	TC implementations — the evidence base of the set version	All TC norms of the version have an admitted implementation with <code>last-run.result = pass</code> and <code>last-run.set-version = N.M</code> ; both TCs of every pair passed; the mandatory TC kinds per SPEC type (§9.8) passed	Verification record in the version record (§10.5.4); for a task — TR <code>done</code>

Checking that `last-run.set-version` matches the set version being verified on every subsequent run is a runner-managed consistency check (§10.9.3), **not** a separate Quality Gate.

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 <code>exp = now + 24h ± 1m</code> "
"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 <code>{\"field\":\"email\",\"code\":\"invalid_format\"}</code> "

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 / set-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
<code>date</code>	yes	ISO-datetime of the run
<code>result</code>	yes	pass fail skipped n/a
<code>runner-id</code>	yes	Runner identifier + version

run-ref	yes	substrate-native pointer to the full run log
set-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 <code>implementation-originated</code> class (§9.18.2); 4) the rest — topped up at random to the sample size
Distribution across types	Uniform across <code>tc-type</code> within the random top-up
Status	Only <code>passing</code>
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
<code>coverage-percent</code> (TC norms of the version with an implementation <code>pass</code> on <code>set-version</code> / all TC norms of the version)	The target threshold is fixed in the conformance manifest	A substrate-native gate blocks promotion
TC norms of the version without an admitted implementation	0 before verification of the version	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
<code>passing-tests</code> / <code>total-tests</code>	100% before change-unit promotion	Blocks QG-2 (§9.10)
<code>manual-pending</code> overdue	0	Notification to the architect; blocking of the affected artifacts

Stale implementations (the norm changed in a version later than <code>automation.set-version</code>)	0	The AI agent re-creates the implementation against the new norm
---	---	---

The two coverage metrics are counted in **different units**, and substituting one for the other is prohibited. `coverage-percent` measures **verification progress** (the share of the version's TC norms with an implementation `pass` on its `set-version`) — the unit of count is the TC norm. 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; that is why QG-0 of the set blocks version approval per assertion (`coverage-presence` , §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 — in the same set draft that will end as version `N.M+1` (§10.5.1):

1. Finds all TC norms whose `verifies[]` point to the artifacts being changed in the draft.
2. Forms a table of affected TC in the frontmatter of the delta-ADAPT or the associated change unit:

TC	Verifies	Action in <code>N.M+1</code>
TC-NN	SR-NN	Norm changed (new step) — the implementation becomes stale (§10.9.4)
TC-NN	SR-NN	No change (still current)
TC-NN	BR-NN	BR removed — the TC norm is removed with it (§10.5.3)

3. Puts the updated TC norms into the same draft as the artifacts derived from the delta-ADAPT; the version is approved by one QG-0 with coverage of every statement (§9.7).
4. After `N.M+1` is approved, implementations of the changed TC norms are created anew against the new norm (`automation.set-version = N.M+1`), pass QG-1 and the run; the former implementations stay in history.

9.16.2 Removal of a TC norm from the set

A TC norm is removed from the next version (§10.5.3) if:

- The verifiable artifact (BR / SR / SPEC) was removed without a replacement;
- The verifiable artifact was replaced by a new one (**replaced-by**) for which a new set of TC norms was written, **and** the old set does not cover the behaviour of the new artifact;
- The behaviour covered by the TC no longer exists in the new version.

A removed TC norm and its implementation are immutable and are 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 norm is approved in a set version and the implementation is admitted to runs (QG-1) before the implementation of the behaviour it verifies is started. A TR MUST NOT enter work while the implementations of the TCs linked to it are not admitted	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 <code>ux / eval</code> to all TC types	The TC provenance (<code>ai-provenance.generated-by</code>) 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 <code>[test-spec-change]</code> 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
<code>id</code>	mandatory	<code>AT-NN</code> ; immutable
<code>verifies[]</code>	mandatory	Only <code>TZ §N</code> and <code>ACTZ-NNN §M</code> . A reference to an internal artifact (BR / SR / SPEC / TC) is fatal
<code>tz-version</code>	mandatory	The revision of the effective TZ from which the AT was derived (§9.19.4)
<code>generator</code>	mandatory	The provenance of the isolated agent: vendor, model, confirmation that it had no access to the internal contour
<code>negative</code>	mandatory	Pos/neg pairing — as for a TC (§9.7)
<code>status</code>	mandatory	<code>draft → ready → passing / failing → obsolete</code> — the same state machine as a TC (§9.9)
<code>environment-ref</code>	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
<code>automation</code>	mandatory	As for a TC, including the mandatory <code>automation.kind</code> (§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
X	✓	Interpretation error : the system conforms to the ADAPT but not to the contract	The ADAPT / ACTZ, then the derived requirements
X	X	Implementation defect	The code
✓	X	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 Manual walkthrough record

9.20.1 Purpose

Automated checking is structurally blind to undescribed behaviour: a TC compares the product with what is **written** and cannot detect behaviour absent from the description. A manual walkthrough of a scenario (SPEC-UC, a SPEC-UI user journey) with the description in hand is the second channel for detecting a divergence of the Source-of-truth drift class (§4.11, row 4.11.3), complementing reconciliation. Its outcome is recorded in a **manual walkthrough record** — an artifact of the evidence class (§7.4.6): the record has no parents or children, is not decomposed, is not verified by TCs and extends no closed list.

9.20.2 Mandatory fields and body

```
id: MW-NN
type: MW
scenario: SPEC-UC-NN | SPEC-UI-NN           # the scenario walked
set-version: "N.M"                          # the set version the behaviour was
checked against
product-version: "<pointer>"                # the product version walked
walked-by: "<actor>"                        # V6; a person other than the
implementation author (P8)
walked-at: "<ISO-8601>"
findings:                                    # zero or more
  - step: 3
    observed: "<what was seen>"
    expected-ref: SR-NN | SPEC-<TYPE>-NN    # the statement it diverges from;
absent if the behaviour is not described
    route: implementation-originated | contractual # §6.13.2: not client-
observable → §6.13; observable → ADAPT / ACTZ
evidence-refs: []                          # pointers to evidence (screenshots,
recordings); the format is not governed (§1.3 item 5)
```

Body: the walk route (scenario steps marked "passed" / "deviation"), findings with a description of what was seen, the conclusion.

9.20.3 Normative rules

1. A manual walkthrough record **does not replace** any TC, **is not part** of the QG-2 evidence base and **is not** grounds for a verification entry of a version (§10.3.3). Complementary manual checking is permitted, substitutive — prohibited (§14.7).
2. Every finding is routed per §6.13.2: behaviour not observable by the client — a candidate **implementation-originated** (§6.13); observable — the contractual contour (a backward finding of ADAPT / ACTZ). A finding without a route is a violation.
3. The walkthrough is performed by a person other than the author of the implementation of the behaviour checked (P8, §9.18).
4. A permanently manual run of **ux** TCs is not provided for (§9.3): perceptual assessment by a person is recorded as a manual walkthrough record, while the TC stays automated.
5. The non-degeneracy measure (§13.9.4): records that systematically yield no finding are a sign of a formal walkthrough; accounted through §12.3.11.

9.21 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 description version through <code>automation.set-version</code> (Statement 3, substrate versioning)
06 Requirements hierarchy	TC verifies BR / SR; <code>verified-by[]</code> — an auto-derived inverse edge on the requirement side
07 ADAPT	TC → SR → ADAPT → TZ — the full trace chain; backward findings (<code>terminology</code> , <code>gap</code>) are fed by patterns from the <code>[test-spec-change]</code> audit (§9.13.3)
08 Specifications	Spec-specific TC per the table in §9.8; SPEC → <code>verified-by[]</code> auto-derived; type-specific extensions §9.6
10 Lifecycle and QG	The TC norm lives in the set version (QG-0), the implementation is admitted to runs (QG-1, §10.9); the verification record of the version (QG-2) requires a run of all implementations; pairing and spec-specific kinds are preconditions of version approval (§10.5.2)
03 Substrate versioning	Immutable IDs (V1); atomic change unit and hooks (V2 + V3); diff & review for <code>[test-spec-change]</code> (V3); TC versioning without loss of history (V4); pinning <code>automation.set-version</code> / <code>last-run.set-version</code> (V5); author + timestamp for <code>last-run</code> (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

10. Lifecycle and Quality Gates

Part of the RENAR Standard v1.1 · ← Table of contents

Dense chapter: read after §6–§9; passing the QGs — guide/00; density — reference/09.

10.1 How the description moves: the set, tasks, gates

The description of the system — BR, SR, SPEC and the normative parts of TC — moves **as a set**, not artifact by artifact. The description set (§4.8.6) is the complete description of the system at the moment of approval; it has one draft and a sequence of approved versions **N.M**. An individual BR, SR, SPEC or TC norm has no state of its own: it is "approved" if it belongs to an approved version of the set, and a "draft" if it exists only in the draft (§10.7). Every approved change of content produces a minor version immediately; a major version is the presented one, approved by the Architect's decision after a full audit (§10.5).

Besides the set, three objects have a lifecycle of their own: the **task** (TR, §10.6) — derived from the set, not part of it; **ADAPT** (§10.8) — the contractual contour; the **TC implementation** (§10.9) — the check code that lives in the code substrate and is admitted to runs separately from the norm.

Moving at will is not allowed: every transition is guarded by a **quality gate** — a condition that **MUST** hold, otherwise the transition does not happen. A set version is not approved while at least one normative assertion is not covered by a pair of TC norms; an ADAPT does not become **approved** while any finding is not closed by a signed clarification protocol; the product is not presented for acceptance while the TCs of the version have not passed on it. A gate is not a "success tick" but a check entitled to say "no" and leave the object where it is.

This chapter brings together the state machines of the four objects and governs the gates: what is checked before each transition, who is obliged to check and when. The chapter fixes **only** states, transitions and gates; artifact frontmatter is defined by chapters 6–9, the set version record — by §10.5.4.

10.1.1 Decision tree: which gate now (informative)

```
flowchart TD
    S[Change] --> T{What transitions?}
    T -->|set draft → version N.M| Q0[QG-0 set approval]
    T -->|task draft → approved| Q0T[QG-0 task approval]
    T -->|TC implementation → admission to runs| Q1[QG-1 check implementation]
    T -->|set version × product version| Q2[QG-2 verification]
    T -->|task approved → done| Q2T[QG-2 of the task]
    T -->|architecture sign-off| Q3[QG-3 optional]
    T -->|business outcome / task acceptance| Q4[QG-4 optional]
    Q0 --> V0{preconditions §10.5.2}
    Q1 --> V1{§10.9.2}
    Q2 --> V2{all TC pass on N.M §10.3.3}
    V0 -->|fail| X[Set stays a draft]
```

```
V1 -->|fail| X1[Implementation not admitted]
V2 -->|fail| X2[No verification record]
```

10.2 Normative definition of a Quality Gate

10.2.1 Quality Gate

Quality Gate (gate) — a normative condition whose check **MUST** be performed for a permitted transition of the **gate object** — the description set, a task (TR), an ADAPT or a TC implementation — from one lifecycle state into another. Every gate consists of:

1. **Identifier** — **QG-N** (closed list §10.3, §10.4). One identifier applies to several objects with different preconditions.
2. **Precondition** — a set of verifiable statements about the object and related artifacts that **MUST** be true at the moment the gate is invoked.
3. **Postcondition** — the state the object transitions into after passing the gate, and the observable effects (for example, a record in the transition log §10.13).
4. **Trigger** — who or what initiates the gate check (a participant: AI agent / Architect / automated runner; an event: approval / run completion / arrival of a delta-TZ).
5. **Enforcement 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**. Passing a gate **MAY** be negative (the precondition is not met) — in that case the transition is forbidden and the object stays in its current state.

10.2.2 Who is obliged to check a gate

Gate type	Mandatory participant	Substrate enforcement
Approval (QG-0)	Architect or an authorized role-holder	Atomic recording of authorship and time (V6, §3.3.6); for the set — atomic release of the version (V2, §3.3.2)
Check implementation (QG-1)	Automated runner (CI, eval-runner)	Atomic recording of the run result pinned to the set version (V5, §3.3.5)
Verification (QG-2)	Automated runner confirming set-version	V5 + V6
Architecture (QG-3, optional)	Architect's signature (ADAPT / architectural part of the set version)	V3 + V6
ACTZ signing	Bilateral signature (client + vendor)	V3 + V6
Acceptance (QG-4, optional)	Stakeholder with authority; for a task — a person other than the executor	V6

10.2.3 Relation to SENAR

SENAR §8 describes Quality Gates as an abstract concept for AI-driven development. RENAR **extends** SENAR in the requirements engineering domain:

- Keeps the identifiers QG-0 / QG-1 / QG-2 as mandatory.
- Governs **formal state machines** for the description set, the task, ADAPT and the TC implementation (SENAR does not).
- Binds every transition of a state machine to a specific gate with preconditions and postconditions.
- Adds the 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 — only the optional §10.4 or through the formal standard change procedure §10.10. Each gate has its own transition object; the full precondition lists per object are in the state machine sections (§10.5, §10.6, §10.8, §10.9).

10.3.1 QG-0 — approval gate

Purpose: permits an object to transition from draft into an approved state: the set draft — into version `N.M` ; a task — into `approved` ; an ADAPT — into `approved` (§10.8).

Precondition for the set (full list — §10.5.2):

- The frontmatter of every artifact of the set is valid against the schema of its chapter; identifiers are unique in the substrate (V1, §3.3.1).
- The tree is complete: every SR has a `parent` , every SPEC has a source. `source.adapt` , if present, points to an ADAPT in a state no lower than `approved` ; otherwise `source.adversarial-review-ref` is present.
- An adversarial review has been performed on every artifact changed relative to the previous version; non-applicability is permitted **only** for trivial changes (by the criteria declared in the conformance manifest, §13) with the reason recorded in the transition log (§10.13).
- Mandatory bodies are filled (§6.5.3, §6.6.3, §8.4.1); for a system-level set SPEC-ARCH and SPEC-SEC are present and non-empty.
- For every normative assertion describing observable behaviour there is a positive + negative pair of TC norms (§9.7, §13.3.5).
- The Architect's signature (V6).

Precondition for a task — §10.6.2; **for an ADAPT** — §10.8.3.

Postcondition:

- Set: the version record `N.M` is released (§10.5.4); the previous approved version transitions into `superseded` ; task planning and TC implementation against `N.M` are permitted.
- Task: `approved` , the executor MAY start. ADAPT: `approved` (§10.8).
- A record in the transition log (§10.13).

Trigger: explicit approval by the Architect / role-holder through the substrate's native mechanism (V3 diff & review, §3.3.3).

Applicable objects: description set, TR, ADAPT.

10.3.2 QG-1 — check implementation gate (TC only)

Purpose: confirms that for a TC norm of an approved set version there is a valid implementation — code, configuration, an infrastructure artifact — fit for running.

Precondition:

- `automation.set-version` points to the approved set version containing this TC norm (V5, §3.3.5).
- `automation.status: automated` (with a valid `automation.location`) or `automation.status: manual-pending` (with `manual-pending-until` and `manual-pending-reason` stated).
- All static checks of the substrate agent's implementation (types, lint, schema) have passed; the runner dry-run has passed.
- The fixing red run is recorded in `red-history` (§9.18.2) — a record of a run before admission, not a state transition of the implementation (§10.9.3); or `not-applicable-reason` is set together with `mutation-check`.
- All mandatory TC body sections (chapter 9 §9.4) are filled.

Postcondition:

- The implementation is admitted to production runs (§10.9).
- A record in the transition log.

Trigger: the approving participant upon the automated runner's confirmation that the dry-run passed.

Applicable objects: TC implementation.

Note: a TR does not pass QG-1 separately: the validity conditions of the task's implementation are part of the QG-2 preconditions (§10.6.2). For BR / SR / SPEC there is no intermediate gate — they are approved as part of the set and verified by its version.

10.3.3 QG-2 — verification gate

Purpose: confirms that the observable behaviour of the product corresponds to the **set version**: all TCs of version `N.M` have passed on it. For a task — that its ACs are confirmed.

Precondition for a set version:

- Every TC norm of version `N.M` has an admitted implementation (QG-1) with `last-run.result = pass` and `last-run.set-version = N.M`.
- Pos/neg pairing is a property of the approved version (QG-0); QG-2 checks that **both** TCs of every pair have passed.
- **Red history** (§9.18.2): every TC has a recorded "red → green" transition. A TC never observed red is not evidence and does not close QG-2. Exception — a TC of the `implementation-originated` class (§6.13): for it a **killed mutant** is mandatory.
- All mandatory spec-specific TC kinds for every SPEC type of the version are present and have passed (chapter 9 §9.8); for SPEC-UI / SPEC-AI — with `judge-isolation` observed (chapter 9

§9.13.4); for SPEC-SEC — a TC `tc-type: security` is present and has passed.

Precondition for a task — §10.6.2.

Postcondition:

- A verification record appears in the set version record (§10.5.4): product version, date, runner, run references. Only the runner writes it, like `last-run`. The state of the set does not change: verification is evidence about the pair (set version, product version), not a state of the description.
- The task transitions into `done`.
- A record in the transition log with evidence-refs (the list of run IDs).

Trigger: the automated runner upon completion of the run of all TCs of the version; for a task — at the executor's request when its TCs pass.

Applicable objects: set version (paired with a product version), TR.

10.4 Gates beyond the mandatory core

This section describes two different things, and they must not be confused.

Optional QG-3 and QG-4 (§10.4.1, §10.4.2) — normatively described but **not mandatory** for conformance (chapter 13). An implementation MAY declare in the conformance manifest either support of QG-3 / QG-4 or their absence. Conformance without QG-3 / QG-4 remains valid.

The AT release gate (§10.4.3) — **mandatory**, not optional. It is **not part** of the closed list of Quality Gates (§10.10.1) and does not extend it: QGs govern the transition of **the set, a task, an ADAPT and a TC implementation** between states, whereas the AT release gate governs the presentation of 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` into `approved` (§10.8). Also applicable to the architectural part of a set version (SPEC-ARCH) 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 `decided-in` pointing to a clause of a **signed** ACTZ (chapter 7 §7.13).
- The Architect's signature is ready (chapter 7 §7.5). The client's signature on the ADAPT is **not required**: client obligations are carried by the ACTZ.
- For a set version (if QG-3 is applied): the decomposition decision is recorded in the substrate as an ADR-like artifact referenced from SPEC-ARCH (the ADR form is substrate-specific — moved to `guide/`).

Postcondition:

- The ADAPT transitions into `approved` (immutable on a par with the TZ).
- For a set version: the Architect's signature on the architectural part is recorded in the version record (§10.5.4).
- 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 bilateral 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 separating it into a distinct gate).
- The architectural part of a set version — in projects with regulatory requirements for architectural acceptance.

10.4.2 QG-4 — acceptance gate (optional)

Purpose: records the stakeholder's acceptance of the business outcome after release — per BR of the presented set version. In implementations where a task is accepted separately from its execution, QG-4 also applies to the task: `done` → accepted by a person other than the executor.

Precondition:

- The set version is verified (QG-2 for the delivered product version) and presented (major, §10.5.1).
- The measurable business outcome (`business-outcome` in the BR frontmatter) is measured by the method declared in `measurement-method` ; the measured value is recorded in the version record (`accepted-outcomes[].measured-value` , §10.5.4), not in the BR — a BR of an approved version is immutable.
- $\text{achievement} = (\text{measured} - \text{baseline}) / (\text{target} - \text{baseline}) \geq$ the project-configurable threshold (80 % by default, fixed in the conformance manifest).
- Formal stakeholder signature; for a task — the signature of an acceptor other than the executor.

Postcondition:

- The acceptance of the BR is recorded in the set version record (`accepted-outcomes[]` , §10.5.4); withdrawing an acceptance requires a delta-TZ.
- A record in the transition log with the stakeholder's signature.

Trigger: formal acceptance by the stakeholder upon release.

When to apply:

- Projects with explicit recording of post-release outcomes (product SaaS, regulated industries).
- Without QG-4 — acceptance of the business outcome is not recorded in the version record; the product's conformance to the contract is proven by the AT release gate.

10.4.3 AT release gate

Purpose: the product is not presented for delivery-acceptance until its conformance to the **contract** is proven. In the first-party confirmation kind (§1.4.4) there is no contract and the gate does not apply: readiness is presented by the verification record of the version (QG-2, §10.5.4).

Precondition:

- All ATs (§9.19) are in status `passing` .
- The ATs are derived from the **effective** revision of the final TZ (§7.14): the `tz-version` of every AT matches the current revision. An outdated trial programme is **not admitted** to trials (§9.19.4).
- The provenance of the AT generator confirms isolation: the model differs from the model of the main agent, there was no access to the internal contour (§9.19.2).

- The presented set version is verified (QG-2) for the delivered product version.

Postcondition: the product MAY be presented for trials; the `ACCEPTANCE.md` report is up to date.

Trigger: the isolated AT generator agent + the automated runner.

The gate is **mandatory** and is not merged with QG-4 (§10.4.2): QG-4 is optional and measures the business outcome, the AT release gate — conformance to the contract. A business goal can be achieved while the contract is violated, 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; objects MAY pass it, but conformance does not require passing it for all. **absent** — the gate is not applied in the implementation; acceptance of the business outcome is not recorded in the version record.

10.5 Description set state machine

10.5.1 States and transitions

```
draft —[QG-0]→ approved N.M —[QG-0 of version N.M+1]→ superseded
    ^                               |
    |                               |
    | new draft ← (any change of the content of an approved
    | version)
```

State	Semantics	Transition gate
<code>draft</code>	The single draft of the set; separated from the approved truth by substrate means (V4, §3.3.4). Contains all artifacts of the current version plus the proposed changes	— (created on the first change after approval)
<code>approved</code>	Approved version <code>N.M</code> ; content is immutable; tasks are planned, TC implementations written and verification run against it	QG-0 (§10.3.1) with the preconditions of §10.5.2
<code>superseded</code>	A version followed by a released successor; kept together with its verification record (V1)	Automatically upon approval of the next version

The **major** flag distinguishes a **major** (presented) version from a **minor** one:

- A **minor version** **N.M** is released on **every** approved change of content — adding, changing or removing an artifact — immediately. No decision "is this change large or small" is taken: any change yields a version.
- A **major version** **N+1.0** is released by the Architect's decision at a milestone (end of a stage, delivery) and only after a **full audit** of the set (§10.5.2). QG-2 for the delivered product and the AT release gate (§10.4.3) are performed against the major version; the stakeholder need only follow major versions.

History is linear: every version has exactly one predecessor (**supersedes**) and at most one successor; there is exactly one draft of the set. Parallel unfinished versions are not permitted — parallel work on independent parts of the description is done in the one draft and approved as one version. The implementation branches, the description does not.

Every change of the content of an approved version — including applying a delta-ADAPT (chapter 7 §7.6) or an accepted change concept — creates a draft and ends with the next minor version. An approved version has no separate "edit in place" path.

10.5.2 Version approval preconditions (QG-0 of the set)

Common part — §10.3.1. The full list for version **N.M** :

Group	Precondition
Structure	The frontmatter of every artifact is valid; identifiers are unique (V1); mandatory bodies are filled (§6.5.3, §6.6.3, §8.4.1, type-specific §8.5); for a system-level set SPEC-ARCH and SPEC-SEC are present and non-empty
Tree	Every SR has a <code>parent</code> in the same version; every SPEC has a <code>source</code> ; the <code>depends-on[]</code> graph is acyclic (§8.6.3), all SPECS from <code>depends-on[]</code> belong to this version; the <code>implements[]</code> of a subsystem BR is valid (§13.3.8)
Provenance	<code>source.adapt</code> , if present, points to an ADAPT in a state no lower than <code>approved</code> ; otherwise <code>source.adversarial-review-ref</code> is present; every <code>approved</code> ADAPT is reflected in the set (§7.4.1)
Review	An adversarial review has been performed on every artifact changed relative to N.M-1 (for the first version — on all)
Coverage	For every normative assertion describing observable behaviour there is a positive + negative pair of TC norms (§9.7, §13.3.5); the mandatory TC kinds for every SPEC type are present (§9.8)
Removals	For every removed artifact the conditions of §10.5.3 are met
Signature	The Architect's signature (V6); in the first-party confirmation kind (§1.4.4) — the Architect's and the responsible person's

Full audit (mandatory for a major version, for a minor one — by the Architect's decision): the preconditions of the table are checked over the **whole** set, not only over the changed artifacts; additionally — consistency of statement forms (§6.6.3.1), absence of contradicting assertions across artifacts, currency of numbering and cross-references, absence in the set of artifacts sourced from a **superseded** ADAPT. The audit outcome is recorded in the version record (**audit** , §10.5.4).

Postcondition: the version record `N.M` is released; `N.M-1` → `superseded` ; the draft holds no changes relative to `N.M` until the next change.

Trigger: explicit approval by the Architect through the substrate's native mechanism (V3 + V6).

10.5.3 Removing an artifact from the set

An artifact does not "transition into `deprecated` " — it is **removed** from the next version while staying in history (V1). The removal is recorded in the version record (`changes[ ].removed`) and approved together with the version.

Precondition:

- `replaced-by` (if there is a replacement) points to an artifact belonging to the same version.
- There are no active tasks (TR in `approved`) on the removed artifact: tasks are redirected to the replacement or moved into `obsolete` (§10.6.3) atomically with the version approval (V2).
- TC norms that verified **only** the removed artifact are removed in the same version; their implementations are withdrawn from runs (§10.9.4).
- Child artifacts (SRs of a removed BR, SPECs of a removed SR) have a new parent or are removed together with it.

Postcondition: the artifact is absent from `N.M` ; the version record carries `removed: [{ id, replaced-by }]` ; the artifact's history and earlier versions remain recoverable (V1).

Trigger: the Architect or the Product Owner — as part of the version approval.

10.5.4 Set version record

Approving a version releases a **version record** — a substrate artifact with the fields:

```
set-version: "N.M"                # identifier; immutable after release
major: boolean                    # true – the presented version (§10.5.1)
approved-by: "<actor>"              # Architect (V6)
approved-at: "<ISO-8601>"
supersedes: "N.M-1"              # exactly one predecessor; absent on the
first version
resolves-to: "<pointer>"          # substrate-native way to resolve the version
into content:                    # a snapshot, a tag, a list of (artifact-id,
                                # version-id) pairs per V5
changes:
  added: [BR-NN, SR-NN, SPEC-<TYPE>-NN, TC-NN]
  changed: [SR-NN]
  removed: [{ id: SPEC-<TYPE>-NN, replaced-by: SPEC-<TYPE>-MM }]
audit:                            # mandatory when major: true
  full: boolean
  findings-ref: "<pointer>"
architecture-signoff:            # if QG-3 is declared and applied to the
version
  signed-by: "<actor>"
  signed-at: "<ISO-8601>"
verification:                    # bot-managed; runner only (§10.3.3)
```

```

- product-version: "<substrate-native pointer>"
  date: "<ISO-8601>"
  result: pass | fail
  runner-id: "<runner-name@version>"
  evidence-refs: [<run IDs>]
accepted-outcomes: # QG-4, if declared (§10.4.2)
- br: BR-NN
  measured-value: number # the KPI value per the BR's measurement-
method
  measured-at: "<ISO-8601>"
  achievement: 0.87 # (measured - baseline) / (target -
baseline)
  accepted-by: "<actor>"
  accepted-at: "<ISO-8601>"

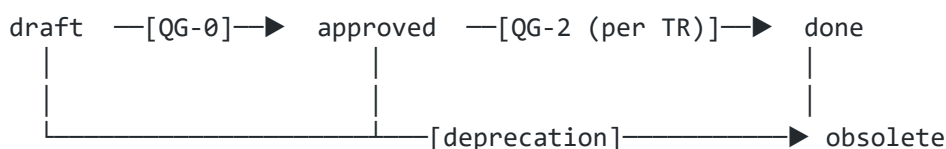
```

Normative rules of the record:

1. The **set-version** identifier and the sections **changes**, **resolves-to**, **audit**, **approved-*** MUST NOT change after release (V1). The sections **verification**, **accepted-outcomes** and **architecture-signoff** are append-only evidence: **verification** is written only by the runner (§9.12 *mutatis mutandis*), **accepted-outcomes** — by the stakeholder with a signature (V6).
2. **resolves-to** MUST resolve into the content of every artifact of the version unambiguously and reproducibly; the pointer form is substrate-specific and is governed in **guide/**. A list of **(artifact-id, version-id)** pairs is mandatory only where the substrate provides no atomic addressable snapshot of a group (V2 without a transaction identifier).
3. All references to a version of the description — **automation.set-version** and **last-run.set-version** of a TC implementation (§9.3), the base point of a delta-ADAPT (§7.6), the conformance manifest (§13.4.2) — point to **set-version**, not to versions of individual artifacts.

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 passed on the set version the task was planned against	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` ; the parent artifact belongs to the approved set version the task is planned against.
- 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` with `last-run.set-version` equal to the task's set version).
- 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 is removed from the next set version (§10.5.3) or changed in it 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 Set artifacts: derived state

10.7.1 State is derived from version membership

BR, SR, SPEC and the normative parts of TC have `no status` and `version` fields of their own. Their state is derived from membership in set versions (§10.5) and is computed by the substrate, not stored:

Derived state	Condition
draft	The artifact (or its change) is present in the set draft and absent from the current approved version

approved	Belongs to the current approved version N.M
removed	Belonged to an earlier version and is absent from the current one; <code>changes[].removed</code> carries <code>replaced-by</code> if there is a replacement (§10.5.3)

Consequences:

- A reference to an artifact (`parent` , `constrained-by[]` , `implements-spec[]` , `verifies[]` , `depends-on[]`) resolves **in the context of a set version**: "the referenced artifact is approved" means "belongs to the same version". A reference from an approved version to a removed artifact violates the preconditions of §10.5.2.
- "Verified" is not a state of an artifact but a property of the pair (set version, product version) (§10.3.3): the same SR is verified on one product version and not verified on another.
- The change history of an artifact is a property of the substrate (V1), not a frontmatter field; comparing two states of an artifact is a diff between set versions (V3).

10.7.2 Structural completeness

The structural completeness check — formerly the SPEC transition into `review` — is a **precondition of version approval**, not a state of its own. Before approval the substrate MUST check for every artifact of the draft that:

- all mandatory frontmatter fields of its chapter are filled (§6.5.2, §6.6.2, §8.4, §9.3);
- all mandatory body sections are present (§6.5.3, §6.6.3, §8.4.1, §9.4);
- the type-specific SPEC sections (§8.5) are present for the respective `spec-type` ;
- the SPEC-ARCH screen list is covered by SPEC-UIs of the same version, and `screens[]` of every SPEC-UI stays within the list (§8.5.6.1);
- from the `RENAR-2` level — the normative sections of artifacts observe the rules of the description language (§15.2–§15.5); a violation is returned as a remark on the wording (§15.7).

On violation the substrate MUST return the list of missing fields and sections (V3 diff & review supports structural feedback); the version is not approved.

10.7.3 Preconditions by SPEC type

In addition to §10.5.2, when approving a version that contains SPECs:

- A version of the `system` level contains non-empty `SPEC-ARCH` and `SPEC-SEC` (chapter 8 §8.3).
- The `depends-on[]` graph is acyclic (chapter 8 §8.6.3); all SPECs from `depends-on[]` belong to the same version.
- If a SPEC references an ADAPT via `source.adapt` — the ADAPT is in state `approved` .
- The mandatory TC-norm kinds per SPEC type are present (chapter 9 §9.8): for SPEC-AI — `evaluation-criteria` fully covered by pairs; for SPEC-SEC — `tc-type: security` ; for SPEC-DATA — `tc-type: contract` for published interface fields; for SPEC-UI — `tc-type: ux` .

Passing these TCs is the subject of QG-2 of the version (§10.3.3); their presence — the subject of QG-0.

10.8 ADAPT state machine

10.8.1 Macro-states



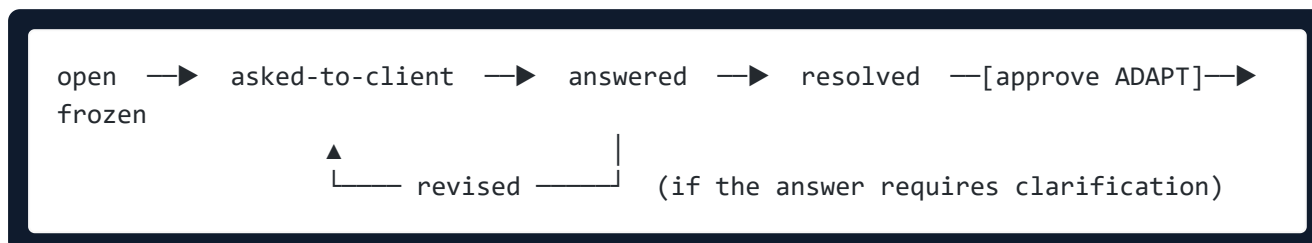
The ADAPT states (**draft** → **review** → **asked** → **answered** → **approved** → **frozen** , and the terminal **superseded** upon supersession) are defined in [chapter 7 §7.4](#) and [§7.6.4](#). The state **client-ready** is withdrawn: putting questions to the client is the subject of an ACTZ ([§7.13](#)), 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 **frozen** and from **obsolete**, the state through which a TR becomes outdated; 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).

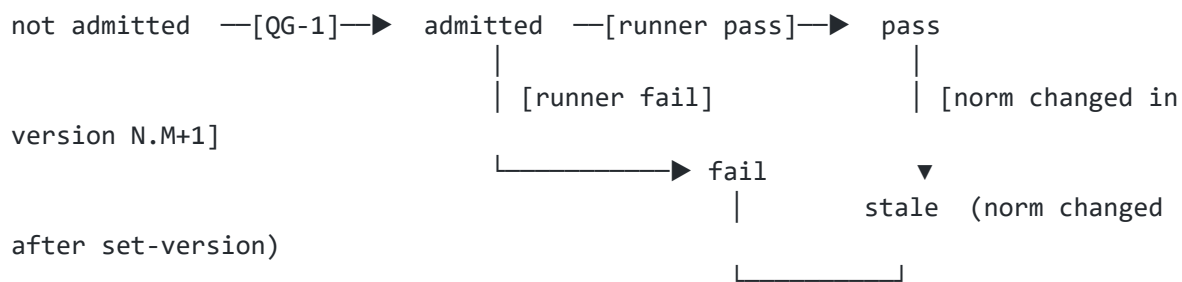
Trigger: approval of the superseding ADAPT via QG-3.

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 implementation: admission and run states

The normative part of a TC is an item of the description set and has no state of its own (§10.7). A lifecycle of its own belongs to the **implementation** of the TC — the check code addressed by `automation.location` and living in the code substrate (chapter 9 §9.17).

10.9.1 States and transitions



State	Semantics	How it is recorded
not admitted	The implementation is absent or has not passed QG-1	<code>automation.status</code> not confirmed by a dry-run
admitted	The dry-run passed; the red history is recorded; production runs are permitted	QG-1 (§10.3.2) with the preconditions of §10.9.2; <code>automation.set-version</code> = version of the norm
pass / fail	Result of the last run on version <code>last-run.set-version</code>	runner; bot-managed (§9.12)
stale	The TC norm was changed in a later set version; a run of the implementation against the previous norm is not evidence	An approved version later than <code>automation.set-version</code> exists whose <code>changes[].changed</code> contains this TC norm (§10.9.4)

TC frontmatter and pos/neg pairing are defined in chapter 9. Removing a TC norm from the set — §10.5.3: the implementation is withdrawn from runs, history is kept.

10.9.2 Preconditions for a TC implementation

QG-1 (admission to runs) — in addition to the common part §10.3.2:

- `automation.set-version` points to an approved set version containing this TC norm, and no later approved version changed that norm (`changes[].changed`).

- `automation.status: automated` (with a valid `automation.location`) **or** `automation.status: manual-pending` (with `manual-pending-until` $\leq +1$ sprint and a filled `manual-pending-reason`).
- The runner dry-run passed (structural validity only; not to be confused with a production run).
- The fixing red run is recorded (§9.18.2); the implementation author differs from the author of the implementation of the behaviour being checked (P8).
- All mandatory TC body sections (§9.4) are filled.

Postcondition:

- The implementation is admitted; the runner is permitted to run in production.
- A record in the transition log.

10.9.3 Runner-managed transitions (not Quality Gates)

`admitted → pass`, `admitted → fail`, `pass → fail`, `fail → pass` — 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 passage of a gate with preconditions and postconditions. In particular, checking that `last-run.set-version` matches the set version being verified (§10.3.3) is a runner-managed consistency check, not a separate gate.

Postcondition of every runner transition:

- `last-run` is updated: `result`, `date`, `set-version`, `run-ref`, `evidence-refs`.
- The substrate **MUST** forbid manual modification of `last-run` (runner-actor only).

10.9.4 Staleness of an implementation

An implementation becomes stale, without entering a terminal state, in two cases:

1. **The TC norm was changed** in set version `N.M+1` (`changes[].changed` contains its identifier): the implementation with `automation.set-version = N.M` is marked stale; its run does not count towards the verification of `N.M+1`. A new implementation is created **against the new norm**, not by editing the old one; the former one stays in history (V1). A delta-TZ invalidating the tested behaviour follows the same path (chapter 9 §9.16).
2. **The TC norm was removed** from the version (§10.5.3): the implementation is withdrawn from runs; nothing is deleted.

The substrate **MUST** detect the first case automatically by the existence of an approved version later than `automation.set-version` whose `changes[].changed` contains the TC norm's identifier (V5) — the enforcement point "TC implementation staleness" (§10.11.1).

10.9.5 Change-of-criteria — a separate normative path

Changing `## Pass criterion` or `## Fail criterion` in a TC norm is **not an ordinary change of set content**; it is a special path requiring a separate approval workflow (chapter 9 §9.13) before approval of the version the change belongs to. Enforcement details — §10.11.3.

10.10 Closed list policy

10.10.1 Normative rule

The closed list of RENAR Quality Gates — mandatory {QG-0, QG-1, QG-2} and optional {QG-3, QG-4}. The list MAY change **only** through the formal RENAR standard change procedure ([chapter 13](#)).

The list closes **Quality Gates** — checks of the transition of **the description set, a task (TR) and an ADAPT** between states and of the **admission of a TC implementation** to runs. The mandatory AT release gate (§10.4.3) is not part of it and does not extend it: it governs the presentation of the product for trials, not a transition of one of the four objects. For the same reason the substrate's transitional events (`freeze` , artifact removal, `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, <code>qg-2.required-negative-tc: true</code>)
Locally weakening the preconditions of a canonical gate	Prohibited	Violates the standard's contract
Declaring QG-3 / QG-4 as <code>absent</code> in the conformance manifest	Permitted	Optional gates — §10.4
Declaring QG-0 / QG-1 / QG-2 as <code>absent</code>	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`).
4. A migration guide for existing conformant projects (§1.7.4, §13.9.3): changes to the self-assessment checklist, new manifest fields.

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 (approval of a set version; a task transition into <code>approved</code> / <code>done</code> ; ADAPT transitions; admission of a TC implementation)	The preconditions of the corresponding gate (§10.3, §10.4, §10.5, §10.6, §10.8, §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)
coverage-presence validation (§13.3.5)	At the approval of a set version: for every normative assertion of BR / SR / SPEC describing observable behaviour, a positive + negative pair of TC norms exists in the same version, addressing it through <code>verifies[].statement</code> (§15.1.1). The presence of coverage is checked, not only the validity of existing <code>verifies[]</code> ; an assertion without a pair — fatal, the version is not approved	V3 (block approve); V1 (the list of assertions of the version is stable)
Reference-validation (any creation/change of an artifact with a reference to another)	The referenced artifact exists: for set artifacts — in the same version (§10.7.1); for ADAPT / ACTZ / AR — 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 of a TC implementation (<code>admitted</code> → <code>pass</code> / <code>fail</code> , §10.9.3)	Only the runner-actor may write <code>last-run</code>	V6 (authorship); the substrate's native ACL or role-based restrictions
TC implementation staleness (TC norm changed in version <code>N.M+1</code> , §10.9.4)	The implementation with <code>automation.set-version = N.M</code> is marked stale; its run does not count towards the verification of <code>N.M+1</code>	V5 (<code>set-version</code>) for detection
implements -edge validation (subsystem BR referencing a	(1) the target BR exists by <code>id + scope.system</code> ; (2) the target belongs to the approved set version of its system at the approval of the version containing this BR (a removed	V1 (stable identifier for target lookup); V3 (block approve

system BR, §6.5.2, §6.8.2)	target — warning, not fatal; cascade-warning over implemented-by[]); (3) the implements chain forms no cycles; (4) implements[] is absent when level: system	until validation passes); V5 (cross-substrate ID resolution when the target is in another substrate)
uses -edge validation (a consumer BR with an edge to a component, §6.14.4)	(1) the component exists by system-id , set-version is an approved version of its set; (2) every A-NN of the component version is confirmed in assumptions-confirmed[] (V6) — unconfirmed — fatal; (3) the uses graph is acyclic; (4) targets in a foreign tree in constrained-by[] — fatal; (5) the removal of a component artifact in a new version — a warning to the consumer, not fatal. The stop rule for incompatible consumer statements (§6.14.4, rule 7) is not computed by the substrate — it is the consumer's obligation and a subject of the adversarial review	V1 (stable identifiers of the component and of assumptions); V5 (the component's set-version); V3 (block approve); V6 (confirmation)
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 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 .	V3 (block approve); V6 (dual signature); V1 (a 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-	V3 (block approve until validation passes); V6 (verdict + signature attribution); V1 (AR immutable once issued)

	findings — no ADAPT, and the BR/SR/SPEC have <code>source.tz-section</code> + <code>source.adversarial-review-ref</code> pointing at that AR. (4) AR integrity: the <code>reviewer</code> model differs from the primary agent's model (§7.10.2); a reference to an AR in status <code>draft</code> or <code>superseded</code> is fatal; a dangling reference is fatal. (5) No mixing: an artifact with <code>source.adapt</code> omitted and without <code>source.adversarial-review-ref</code> — fatal.	
adapt-supersession validation (§7.6.4, §10.8.5)	(1) <code>supersedes</code> : ADAPT-MMM references an existing ADAPT; the back-reference <code>superseded-by</code> is symmetric. (2) <code>supersession-rationale</code> 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 <code>superseded-by</code> on the former one. (4) A dangling <code>source.adapt</code> reference to an ADAPT in status <code>superseded</code> — 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	Object	Why prohibited
draft	superseded	Set	Skips QG-0; a version that never existed cannot be superseded
approved	draft	Set	An approved version is immutable (V1); editing content — a new draft and version N.M+1 (§10.5.1)
superseded	approved	Set	Linear history; returning to earlier content — only as a new version with the same content and <code>supersedes</code> pointing to the current one
approved N.M	approved N.M+2	Set	Skipped version; every version has exactly one predecessor
draft	done	TR	Skips QG-0
obsolete	*	TR	Terminal status; "resurrection" is prohibited — a new task is needed
not admitted	pass / fail	TC implementation	Skips QG-1 (no dry-run); the fixing red run before admission is written to <code>red-history</code> , not to <code>last-run</code>
admitted / pass / fail	not admitted	TC implementation	Degradation loses run history; weakening a check — only through change-of-criteria (chapter 9 §9.13), staleness — §10.9.4
frozen	*	ADAPT	Terminal along the derivation line; changes only via delta-ADAPT or errata (§10.8.4)
superseded	*	ADAPT	Terminal; immutable (§10.8.5)
approved	draft	BR / SR / SPEC / TC norm	An artifact has no transition of its own: a change goes through the set draft and the next version (§10.7)

10.12.1 Substrate reaction

On an attempted prohibited transition the substrate MUST:

1. Block the transition (V3 diff & review).
2. Return to the calling participant an error code naming the specific 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, artifact removal, freeze-transition) MUST be recorded in the substrate as an immutable event with the following fields:

Field	Semantics	Mandatory
timestamp	UTC ISO-8601 moment of the successful pass	Mandatory
artifact-id	Identifier of the object (immutable, V1): set-version for the set, the ID of the task, ADAPT, TC	Mandatory
artifact-type	SET (set version) / TR / ADAPT / ACTZ / AR / TC (implementation) / AT ; for the removal event — the type of the removed artifact (BR / SR / SPEC / TC-NORM)	Mandatory
set-version	The set version the event relates to: for the set — the released one; for a task and a TC implementation — the one they are conducted against (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) / removal / freeze / runner-pass / runner-fail / change-of-criteria	Mandatory
actor	Identifier of the initiator (V6); for the bilateral ACTZ signature — the list of participants	Mandatory
evidence-refs	References to evidence: runner run IDs, adversarial review artifact IDs, signature IDs, a pointer to the full audit outcome	Mandatory for QG-2 / QG-3 / QG-4 and for a major version
notes	Free text	Optional

For a QG-0 event of the set, **evidence-refs** additionally carries the list of artifacts from **changes[]** of the version record (§10.5.4) — one event per version, not per artifact.

10.13.2 Substrate-independent format

The storage format of events is substrate-specific (a separate log stream / an append-only collection / other forms). The standard governs only the mandatory fields of §10.13.1, not their serialization.

The conformance manifest MUST state the event storage mechanism and the export format (for audit, chapter 13).

10.13.3 Retention

Events **are not deleted** throughout the lifecycle of the object and after its transition into **superseded** / **obsolete** / **frozen** , likewise after the removal of an artifact from the set. 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)
04	The terms "description set" and "set version" (§4.8.6); gate terms (§4.7)
06	BR / SR / TR frontmatter and body (§6.5–§6.7); derived state of BR / SR — §10.7; TR state machine — §10.6
07	ADAPT frontmatter (§7.8); backward sub-states (§7.4.5); Architect's signature (§7.5); ACTZ and the bilateral signature (§7.13); a delta-ADAPT produces a set version (§7.6) — state machine here (§10.8)
08	SPEC frontmatter (§8.4); type-specific QG (§8.8); derived state and structural completeness — §10.7
09	TC frontmatter (§9.3); pos/neg pairing (§9.7); change-of-criteria (§9.13); TC implementation — §10.9
03	V1–V6 — the foundation of enforcement (§10.11); the baseline state (§3.3.3) = the approved set version; without V3 / V4 / V6 — no gate implementation
11	Maturity levels define the scope of applicable gates (for example, RENAR-1 — QG-0 / QG-1 / QG-2 mandatory; higher levels strengthen the QG-0 preconditions of the set: pos/neg pairing, spec-specific TC)
13	The conformance manifest declares gate support (§10.4.4) and the set version (§13.4.2); holds the event retention policy (§10.13.3)

11. Maturity Model

Part of the RENAR Standard v1.1 · [← 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 $N \in \{1, 2, 3, 4, 5\}$ – general SENAR maturity

$M \in \{1, 2, 3, 4, 5\}$ – 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 the artifacts of the approved set version have <code>verified-by</code> ; pos/neg pairing; QG-2 is enforced; AI provenance
RENAR-5	Optimized	Adversarial review as a gate; multiple models for <code>priority: must</code> ; 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; the normative sections of artifacts observe the rules of the description language — operators, glossary discipline, prohibited constructions, atomicity (§15.2–§15.5).

11.5.2 What is NOT required at RENAR-2

CI validation of frontmatter against a schema; the states of TR / ADAPT / TC implementations — at the team's discretion; `tc-type` extensions; TC implementations — a norm without an implementation is permitted, the implementation is admitted to runs as it becomes ready (§10.3.2). Pairs of TC norms per statement of the version are not waived: that is the mandatory clause §13.3.5, in force at every level.

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); description set versions are actually released, the states of TR / ADAPT / TC implementations are used ([chapter 10](#)); TC implementations are admitted to runs (QG-1) for all norms of artifacts 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 `automation.set-version` (§9.3, V5).

11.6.2 Observable signals

The frontmatter-validation hook returns a structured response on a schema violation; at least one approved set version has been released (§10.5); TR, ADAPT and TC implementations are in the states of §10.6, §10.8, §10.9; 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 TC norm of the version has an implementation that passed on its `set-version`; the verification record of the version is not created without green tests

on that 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 the TC norms of the approved set version have an implementation with the result `pass` on `set-version = N.M` (§9.9) — the pairing of norms per statement is already given by §13.3.5, the level adds the run; QG-2 (§10.3.3) is **enforced** — recording the verification of a version is blocked unless all TCs have passed on its `set-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 — the `ai-provenance` block in the composition fixed by the canonical §4.10.1 (`generated-by` , `generated-at` , `prompt-template` , `context-tokens` , `output-tokens` , `human-edits`); this section does not redefine that composition; 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 the approved version; an attempt to record the verification of a version without all TCs passed 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	✓	✓	✓
Set versions released; states of TR / ADAPT / TC implementations kept	✗	partial	✓	✓	✓
Frontmatter schema validation (substrate hook)	✗	✗	✓	✓	✓
TC as a full-fledged artifact	✗	✗	partial	✓	✓
An implementation with <code>pass</code> on <code>set-version</code> for every TC norm of the version	✗	✗	✗	✓	✓
COVERAGE auto-generated	✗	✗	✓	✓	✓
Reference-validation hook	✗	✗	✓	✓	✓
<code>automation.set-version</code> 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

What is added between adjacent levels. The numbering is for reference, not a prescribed order of work: the order of adoption is the organization's choice (§1.2.1); the recommended one is [guide/02](#). The times are an expected order of magnitude for a small project, 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. Release the first approved set version (QG-0, §10.5.2); keep the states of TR / ADAPT / TC implementations.
4. Enable the reference-validation hook (§10.11.1).
5. Generate the initial auto-generated COVERAGE artifact (§9.15).
6. Admit to runs (QG-1) the TC implementations for the norms of artifacts with `priority: must` .
7. Introduce the `automation.set-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 writes implementations for all TC norms of the version (the pairs of norms per statement are already in the version per §13.3.5).
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 (the verification record of the version is blocked without `pass` of all implementations on its `set-version`).
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 12 SPEC types — mandatory; full type-specific coverage at RENAR-4+; continuous evaluation of SPEC-AI at RENAR-5
09 Test cases	TC for <code>priority: must</code> 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 v1.1 · ← 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 approval of the first description set version containing the whole BR/SR tree of this TZ (QG-0 of the set, §10.5.2). **Formula:** $RDLT =$

$\text{timestamp}(\text{set-version with the full TZ tree} \rightarrow \text{approved}) - \text{timestamp}(\text{TZ registered})$. 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 approval of the set version in which the SR is first approved to the creation of the first TR with the reference **implements: SR-N**. **Formula:** $\text{Latency} = \text{timestamp}(\text{first TR.created}) - \text{timestamp}(\text{first set-version} \ni \text{SR} \rightarrow \text{approved})$. 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:** $\text{assertions_without_valid_citation} / \text{total_normative_assertions} \times 100\%$. 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:**

$\text{BRs_with_high_disagreement} / \text{total_must_BRs} \times 100\%$. **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:** $\text{disputed_BRs_at_QG4} / \text{total_BRs_in_release} \times 100\%$. **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:**

$$\text{artifacts_with_critic_high_findings} / \text{total_reviewed_by_critic} \times 100\%$$

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. In the first-party confirmation kind (§1.4.4) the metric is mandatory at any level: the adversarial review is the only independent check there, and the ACR is the measure of its non-degeneracy (§13.9.4).

12.3.7 Test-spec Drift Rate

The percentage of TC implementations with `last-run.result = pass` (§10.9.1) whose

`automation.set-version` (§9.12) is earlier than the set version in which their norm was last changed (§10.9.4). **Formula:**
$$\text{stale_passing_TCs} / \text{total_passing_TCs} \times 100\%$$
 (stale = the implementation is stale). **Source:** the COVERAGE artifact (§9.15). **Relationship to SENAR:** a new metric. **Targets:** RENAR-1..3 n/a; RENAR-4 $\leq 5\%$; RENAR-5 $\leq 1\%$ (auto-rerun on delta-ADAPT).

12.3.8 Coverage Velocity

The share of TC norms of the current set version whose implementations passed on it (`last-`

`run.set-version = N.M` , §10.3.3), gained per unit of time (default iteration; the substrate MAY define a different interval in the manifest). **Formula:**
$$(\text{passed_on_version}(\text{end}) - \text{passed_on_version}(\text{start})) / \text{tc_count}(N.M) \times 100\%$$
 . **Source:** the COVERAGE artifact history (§9.15). **Relationship to SENAR:** a refinement of `Throughput` for requirements. **Targets:** RENAR-3 $\geq 30\%$ /iteration; RENAR-4 $\geq 50\%$ /iteration; RENAR-5 $\geq 70\%$ /iteration.

12.3.9 Cost per Approved Requirement

The AI-generation cost (input tokens + output tokens $\times$ tariff) normalized to one artifact that first entered an approved set version, including rejected drafts (which remain in the substrate by virtue of V1 immutable history). **Formula:**
$$\text{total_AI_tokens_cost}(\text{period}) / \text{count}(\text{artifacts_first_approved_in_period})$$
 . 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:**

$$\text{count}(\text{reconciliation_findings_registered}) / \text{weeks_in_period}$$
 . **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:** $\text{IOR} = \text{count}(\text{SR where source=implementation-originated}) / \text{count}(\text{all SR added}) \times 100\%$. **Source:** the **source** frontmatter field + the audit log. **SENAR mapping:** a new metric. A zero value over a period in the presence of manual walkthrough records (§9.20) is a sign of a formal walkthrough, not of the absence of findings (after the pattern of §12.3.10). **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.

12.3bis Degeneracy measurers for mandatory controls (informative)

This section **introduces no metrics** and does not extend the closed list of §12.3. It collects what already applies: which of the eleven metrics shows that a given mandatory control is being executed **formally**. A control executed formally yields a degenerate value — zero findings, zero objections, a hundred per cent agreement — and by every other sign looks healthy: the signature is there, the record is issued, the gate is green.

Mandatory control	Degeneracy measurer
QG-0 — approval (§10.3.1)	ACR §12.3.6 — the critic finds ≥ 1 high-severity issue before QG-0; Hallucination Rate §12.3.3 — statements with no traceable source passed the gate
QG-2 — verification (§10.3.3)	Test-spec Drift Rate §12.3.7 — TC in passing against a stale set version
Adversarial review (§7.10.2)	ACR §12.3.6 — values below 20% indicate a weak critic
ACTZ dual signature (§7.13)	DRA §12.3.5 — the client disputes at acceptance what they signed; applicable only when QG-4 is declared
Drift-detection hook (§2.3.3)	Reconciliation Findings/Week §12.3.10 — zero means the hook is not working
The implementation-originated class (§6.13)	IOR §12.3.11
Version pair coverage coverage-presence at QG-0 (§10.11.1, §13.3.5)	Test-spec Drift Rate §12.3.7 — the pairs exist, but the implementations were run on a previous set version; substantive degeneration (a pair written for the count) is the same gap as pos/neg pairing below

Screen-list coverage (§8.5.6.1)	Reconciliation Findings/Week §12.3.10 — class 4.11.3 findings about screens: zero with a live interface means the list was copied from the existing SPEC-UIs rather than derived from the description
Confirmation of component assumptions in the <code>uses</code> edge (§6.14.4)	ACR §12.3.6 over the <code>assumptions[]</code> section — not one assumption contested by the adversarial review with a non-empty list; DRA §12.3.5 — the consumer's client disputes behaviour "confirmed" by an assumption
Description-language rules (§15.7)	Multi-model Disagreement Rate §12.3.4 — formally conforming text is read by models as differently as free prose; ACR §12.3.6 — zero wording remarks at review
Non-empty SPEC-ARCH and SPEC-SEC at the <code>system</code> level (§8.3)	ACR §12.3.6 over these two types — zero findings means a document written for presence; Hallucination Rate §12.3.3 — SPEC-SEC statements without a traceable source

For QG-1, the Architect's signature on an ADAPT, the AT release acceptance gate, the red history of a TC, and pos/neg pairing (including its version form, `coverage-presence`) there is **no** measurer in force. This is a recorded gap, not an omission: it should be closed case by case, each on its own analysis, rather than by a blanket clause.

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, not as statistically validated values. The clause covers the seven metrics that carry a number in the table: RDLT, Requirement-to-Task Latency, Hallucination Rate, DRA, Test-spec Drift Rate, Coverage Velocity, Implementation-originated Rate. It does not cover metrics whose value is **n/a** , **tracked** , or **optional** — there is nothing there to calibrate.*

Calibration condition. *The threshold of a given metric becomes subject to calibration once reported data for that metric have accumulated from no fewer than **five independent adopting organizations** (independent = a separate legal entity; several projects of one organization count as one), each contributing no fewer than one **completed cycle** (from TZ import to the verification record of the version at QG-2 for the artifacts the metric covers; for DRA — to QG-4), with the data traceable to a published conformance manifest (§13.4). The quorum is procedural, not statistical: it separates calibration from conjecture, but grants no basis for claiming statistical validity.*

Who declares the condition met. *Only the formal change procedure of the standard (§13.9): the standard team publishes a calibration report stating the data set, the sources, and the number of organizations and completed cycles per metric, after which the new values enter the corpus through a minor-version bump. Neither an individual nor an adopting organization declares the condition met unilaterally.*

*Until calibration the thresholds **apply as they stand** but are not considered validated: exceeding an uncalibrated threshold does not by itself entail loss of conformance (§13.8.1). See [guide/07 §8.1](#).*

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 set-version pinning.
- Reconciliation Findings per Week (§12.3.10) — specific to continuous reconciliation.
- Implementation-originated Rate (§12.3.11) — specific to the class of requirements originated by the implementation (§6.13).

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 v1.1 · ← 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 <code>frontmatter</code> schema, no lifecycle statuses	Minimum entry level (§13.2.1)
RENAR-2	Documented (fixed in artifacts): a requirements substrate exists; artifacts have a basic <code>frontmatter</code> ; the TZ is recorded	Conformance declarable (§13.2.2)
RENAR-3	Tracked (lifecycle and link accounting maintained): full <code>frontmatter</code> schema; lifecycle statuses are used; delta-TZ workflow through ADAPT	Conformance declarable (§13.2.2)
RENAR-4	Verified: 100% of approved artifacts have <code>verified-by</code> ; 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. **Version-graduated clauses:** some clauses of this section are introduced with a transition period, and their modality depends on the version of the standard the conformance is claimed against; on a version where the clause is stated as a recommendation, failing it does not remove conformance. The current modality is recorded in the machine-readable index of normative points (`reference/normative-index.yaml`). On v1.1 there are no clauses in a transition period: §13.3.8, which was one on v1.0, has been moved to mandatory.

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 pinning <code>set-version</code> — the description set version (§10.5.4) — and QG-2 (§10.3.3) are impossible
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 <code>approved</code> , with the Architect's signature (§7.5). Every finding carries a <code>decided-in</code> pointing to a clause of a signed ACTZ (§7.13). Lifecycle §7.4.5. The AR carries a non-empty <code>produces-adapt[]</code> .	<code>source.adapt</code> mandatory; <code>source.tz-section</code> mandatory
"no findings, no clarifications" — the TZ → RENAR conversion is unambiguous	No ADAPT is created	<code>source.tz-section</code> mandatory; <code>source.adversarial-review-ref</code> mandatory — a reference to an AR with the <code>no-findings</code> 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 12 SPEC types

A SPEC type MUST belong to the closed list of twelve 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` , `SPEC-UC` (the last one added in v1.1 under the procedure of §13.9).

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 of the description set (BR / SR / SPEC) that describes observable behaviour MUST be covered by a positive + negative pair of TC norms in the same set version (§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-0 of the set (§10.5.2) MUST block the approval of a version in which at least one such statement is not covered at all or is covered without a pair. The **coverage-presence** enforcement point (§10.11.1) MUST check the **presence** of coverage, not only the validity of existing **verifies[]** references: a version in which no TC norm at all is written for a statement violates this clause just as a version with a single positive TC does.

Negative scenario: a project declares **RENAR-N** conformance having approved a set version in which an SR with observable behaviour has no TC norm at all — **non-conformant:** the absence of a TC norm is a violation on a par with an incomplete pair.

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 version record carries no accepted-outcomes[] — the verification of the version remains the last evidence

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):

Condition	Requirement
-----------	-------------

BR.level = subsystem AND the parent system has ≥ 1 BR in an approved version of its set	implements[] MUST contain ≥ 1 reference to an applicable BR of the parent system. Its absence when the condition holds is non-conformance.
BR.level = subsystem AND the parent system is a container without BRs of its own	implements[] is omitted; the justification is recorded in the Context section with a reference to ADAPT§ (§6.8.2).

The substrate MUST implement the `implements` -edge validation checkpoint from §10.11.1 (the target belongs to the approved set version of its system, no cycles, removed target $\rightarrow$ warning, `implements[]` not on `level: system`). This obligation applies to the substrate tooling and was already in force on v1.0, when the presence of `implements[]` was still recommended; the checkpoint MUST also verify the **presence** of the edge when the condition holds, not merely the validity of an existing one. Its violation is caught by the checkpoint on change, not through the loss-of-conformance triggers of §13.8.1 — those address violations of mandatory clauses, not a missing check at the substrate.

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 (migration guide).

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

```
# RENAR Conformance Manifest schema (mandatory fields, v1.1)
renar-version: "1.1"           # 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)
set-version: "N.M"             # the approved description set version at
```

```

assessment time ([§10.5.4](10-lifecycle-qg.md#10.5.4))

# Assessment metadata
assessment-mode: "self"           # self | third-party
confirmation: "second-party"      # second-party (a client exists, ACTZ) | first-
party (a concept as the input, §1.4.4; ACTZ and AT have no subject)
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
  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
  implements-edge-subsystem: true # §13.3.8

# 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", "SPEC-UC"]

# All 12 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

- **set-version** — the description set version (§10.5.4) assessed at the declaration; the declaration refers to it, not to individual artifacts.
- **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.
- **confirmation** — **second-party** (the contract-oriented kind §1.4.1: a client, ACTZ, AT) or **first-party** (the kind of §1.4.4: a concept as the input, the ACTZ and the AT release gate have no subject, adversarial review without exception, ACR §12.3.6 mandatory). The value MUST match the actual kind: **second-party** with one person in both ACTZ signatures is a violation of §5.5.3.

- **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–§13.3.8 ceased to hold (for example, a BR appeared without <code>source.adapt</code> and without <code>source.adversarial-review-ref</code> — 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	<code>next-assessment-due</code> 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 calibrated normative threshold for the level (for example, Hallucination Rate > 5% on RENAR-4 — §12.3.3). Exceeding a threshold that §12.4 marks as provisional and that has not yet undergone calibration is not by itself a trigger : conformance is lost either against a calibrated threshold or through any other trigger in this list
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 (<code>RENAR-6</code> , <code>RENAR-PRO</code>)	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 <code>declared-stricter</code>
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 <code>RENAR-3.5</code>	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.

13.9.4 A degeneracy measurer for a new mandatory control

When a new mandatory control is introduced, the standard-change proposal MUST state which measurer determines the degeneracy of that control. A new metric is REQUIRED only when none of the measurers in force (§12.3bis) applies to it.

A control executed formally yields a degenerate value — zero findings, zero objections, a hundred per cent agreement — and by every other sign looks healthy. The requirement addresses the **author of a change proposal**, not projects in operation: it binds future amendments of the standard and has no retroactive force.

The second half of the rule matters no less than the first: it acts as a procedural filter against growth of the closed list of metrics (§12.3). Before proposing a twelfth metric, the author MUST show that none of the eleven in force fits the new control.

The clause does not extend retroactively to controls introduced before it: some of them have no measurer, and that gap is recorded explicitly in §12.3bis.

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 <code>frontmatter</code> 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 12 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 v1.1 · ← 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.set-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 <code>tc-type</code>
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 <code>ai-act.risk-class</code> 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) — substitutive; a complementary manual walkthrough record (§9.20) is permitted and replaces no TC	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	<code>renar-version</code> , <code>external-claims[]</code>
reference/11	Full informative catalog of §14.5

reference/03 AI risk	Risk register per 23894 + NIST
----------------------	--------------------------------

15. Description language

Part of the RENAR Standard v1.1 · ← [Table of contents](#)

*The normative sections of description artifacts — BR, SR, SPEC, TC norms, ADAPT, ACTZ — are written in a controlled dialect of natural language: a closed list of modal operators, glossary discipline, prohibited constructions, atomicity and forms by artifact type. The chapter governs the text of **project artifacts**; the text of the standard itself is governed by the editorial policy of reference/06. Basis — ADR-026; the sources of the techniques are ASD-STE100, EARS, RFC 2119 / 8174, the INCOSE Guide for Writing Requirements, Attempto Controlled English, RuleSpeak, Gherkin (reference/11).*

15.1 Why

The standard governs which artifacts are created and how they are linked ([chapter 6](#), [chapter 8](#), [chapter 9](#)); the statement inside an artifact was until now governed by one form ([§6.6.3.1](#)). Free prose produces four classes of defects that are discovered late — at QG-2 or at the client:

- **breadth of reading** — a phrase admits several readings, the agent picks one with no signal that a choice was made;
- **unverifiability** — "the system MUST be convenient" does not translate into a Pass criterion;
- **terminological drift** — one meaning is named by different words along the chain BR → SR → SPEC → TC;
- **accumulation of ambiguity** along the same chain through `constrained-by[ ]` and `source.*`.

The description language is neither a formal language (no parser, readable by a person) nor free prose: a set of disciplines checked by reading — by a person in the adversarial review and by the agent at generation.

15.1.1 Normative section

The rules of the chapter act in the **normative sections** of artifact bodies: "Need", "Success criteria" and "Constraints" of a BR ([§6.5.3](#)); "Requirement", "Behaviour", "Constraints" of an SR ([§6.6.3](#)); the type-specific sections of a SPEC ([§8.5](#)); the Pass and Fail criteria, preconditions and steps of a TC ([§9.4](#)); the clauses of an ACTZ ([§7.13](#)); the forward interpretation of an ADAPT ([§7.4.3](#)). Free sections — "Context", rationales, version history — are subject only to [§15.2](#) (operators) and [§15.3](#) (terms); correspondence inside an ACTZ (the client's text) is not subject to the chapter's rules.

The address of a statement. A statement is a sentence of a normative section. The statements of one artifact are numbered through its normative sections in order of appearance; the address of a statement is `<id>#<n>` (for example, `SR-10#5`). The numbering is derived from the text and is not stored in it; inserting a sentence shifts the numbers — that is an edit of statements, visible in the version record ([§10.5.4](#)). The address is used by `verifies[ ].statement` of a TC norm ([§9.3](#)), by the `ref` of a scenario step ([§8.5.12.1](#)) and by the version coverage check ([§10.11.1](#)).

15.2 Modal operators (closed list)

Operator	RFC 2119	Semantics
MUST	MUST / SHALL	An absolute requirement; non-fulfilment is a defect
MUST NOT	MUST NOT / SHALL NOT	An absolute prohibition
SHOULD	SHOULD	A default requirement; deviation is permitted with an explicit rationale in the same artifact
SHOULD NOT	SHOULD NOT	A default prohibition; an exception is justified in the same place
MAY	MAY	Truly optional behaviour; no rationale required

The list is closed (§1.7.5, row 17). Three rules:

1. **Uniqueness of form.** In a normative section, obligation is expressed only by an operator from the list; synonyms ("shall", "needs to", "is required to", "it is important that", "it is desirable", "can") are prohibited. The operators are written in upper case per RFC 8174 — the convention the tooling understands (reference/06); the Russian edition of an artifact writes its operators in ordinary orthography, and unambiguity there comes from the closedness of the list and the position, as with the markers of the forms of §6.6.3.1.
2. **One operator per statement.** In one normative sentence — exactly one operator. "MUST, but MAY" is two statements that require separation (§15.5).
3. **"MAY", not "can".** The word "can" is ambiguous between capability and permission ("the system can process 1000 requests" — a capability or a right?) and is not used in normative sections.

The Russian edition of an artifact uses **должен / не должен / следует / не следует / допускается**; the correspondence between the editions is by the table above.

15.3 Glossary discipline

1. **One term — one definition — one form.** Every term that matters for reading a statement (a role, a state, a unit of measure, an entity of the system) is defined once — in the term-mapping section of the ADAPT (§7.4) or at its first use in the set — and is used only in that form thereafter.
2. **Synonyms of one concept are prohibited** along the chain $BR \rightarrow SR \rightarrow SPEC \rightarrow TC$. Two artifacts naming one entity differently ("check" and "validation") are not a stylistic variant but a backward finding of the **terminology** category (§7.4.4); in artifacts originating from the TZ it passes the contractual contour by the general rules.
3. **Units of measure** are mandatory with every numeric value and identical throughout the artifact (always "ms", not "ms" in one place and "seconds" in another).
4. **An abbreviation** is expanded at its first use in the artifact and used only in the short form thereafter; the abbreviations of the standard's closed lists (**BR** , **SR** , **SPEC** , **TC** , **ADAPT** , **ACTZ** , **QG**) require no expansion.

The terms of the standard itself are subject to §4.2; their drift in an artifact is class 4.11.5 (§4.11).

15.4 Prohibited constructions

Every occurrence is reformulated before the set version is approved. The list is closed within the chapter; additions — through a change of the standard.

#	Prohibited	Why	Bad	Good
1	An evaluative word without a criterion: "convenient", "fast", "sufficient", "modern", "intuitive"	Unverifiable	The system MUST work fast	The system MUST return a response within 300 ms
2	An escape clause: "where possible", "as needed", "within reason", "where applicable"	A documented excuse for non-fulfilment	Errors are logged where possible	The system MUST write every error to the <code>errors.log</code> journal
3	The "/" sign as an "and / or" connector	Unclear whether it is a conjunction or an alternative	The field is mandatory/optional	The user MUST fill the field. IF the value is absent, THEN the system MUST substitute <code>null</code>
4	An unattainable absolute without a tolerance: "always", "never", "100 %", "completely"	Literally unverifiable	The system never loses data	The system MUST keep data with a reliability of no less than 99.99 % over a 30-day interval
5	"All / any / both" instead of an element-wise quantifier	Unclear whether the statement is collective or per element	All fields MUST pass validation	Each form field MUST pass validation by its own rule
6	A pronoun without an unambiguous antecedent in the same sentence: "this", "it", "they"	The statement loses self-sufficiency — the agent reads in parts	The user submits the form. It MUST be validated	The system MUST validate the form before submission
7	A negative requirement without a positive measurable criterion: "MUST NOT hang"	Sets no verifiable behaviour	The interface MUST NOT hang	The system MUST display a waiting indicator IF no response is received within 200 ms
8	A connector "and / or / then / unless" joining different conditions or actions	Two requirements glued into one	The system checks permissions and logs the attempt if access is denied	(1) The system MUST check access permissions. (2) IF access is denied, THEN the system MUST write the attempt to the journal
9	Passive voice without the acting subject	Unclear who acts — responsibility is not assigned	The request MUST be approved	The manager MUST approve the request

10	A subordinate clause attached to a word other than the one it refers to	Ambiguous attachment	The user receives a notification about the request that was rejected	IF the request is rejected, THEN the user MUST receive a rejection notification
----	---	----------------------	--	---

The informative word list of [reference/04 §3.2](#) illustrates rows 1 and 2; the norm is here.

15.5 Atomicity of a statement

One normative sentence — one verifiable thought.

The criterion. A statement is not atomic and MUST be split if more than one pair of TC norms — a positive and a negative one (§13.3.5) — is needed to verify it; for a negative invariant — more than one TC norm. Signs: more than one operator in the sentence; "and" joins two actions or two conditions rather than homogeneous objects of one action; the sentence answers two "what must happen" questions at once.

Length. A statement SHOULD fit into 25 words. Exceeding that is a sign of gluing, not a reason to shorten the text at the expense of meaning.

15.6 Forms by artifact type

The forms are a recommendation in v1.1 (SHOULD), with a review of the modality by practice (ADR-022, ADR-026); the rules of §15.2–§15.5 act regardless of the form.

15.6.1 SR and SPEC

Statements about the behaviour of the system in an SR and in the type-specific sections of a SPEC take one of the six forms of §6.6.3.1: ubiquitous, state-driven (WHILE ...), event-driven (WHEN ...), unwanted behaviour (IF ..., THEN ...), optional feature (WHERE ...), complex (WHILE ..., WHEN ...). The word "system" is replaced by the exact name of the component when the statement is addressed to it rather than to the system as a whole (§15.4, row 9).

15.6.2 BR

"Need" takes the form "who, what, why" (§6.5.3). The "Constraints" section of a BR holds business rules in four forms:

Form	Template	Example
Obligation	⟨Subject⟩ MUST ⟨action⟩	The client MUST confirm payment before work begins
Prohibition	⟨Subject⟩ MUST NOT ⟨action⟩	The contractor MUST NOT engage a subcontractor without written approval
Restricted permission	⟨Subject⟩ MAY ⟨action⟩ only if ⟨condition⟩	The client MAY request a refund only if the request is filed within 14 days
Conditional obligation	IF ⟨condition⟩, THEN ⟨subject⟩ MUST ⟨action⟩	IF the amount exceeds 500 000 P, THEN the manager MUST escalate the request

"Only if" is mandatory in a restricted permission: without it, it is unclear whether the condition is mandatory or one of the admissible ones. "Success criteria" — 3–7 measurable atomic statements.

15.6.3 TC

The body of a TC is governed by the seven sections of §9.4; the triple "Given — When — Then" reads as three of them: **Given** → Preconditions, **When** → Steps, **Then** → Pass criterion. The Fail criterion is a list of observable signs of violation, not a negation of Pass (§9.11). For `system` and `contract` the steps SHOULD contain one action or one event; several actions — several TCs. Journey TCs (`ux` , SPEC-UC) are multi-step by construction — the restriction does not extend to them.

15.6.4 ADAPT and ACTZ

The forward interpretation of an ADAPT is prose under §15.2–§15.5; a form on behalf of the analysis ("is interpreted as ...") is permitted. A clause of an ACTZ is an obligation to the client: the form of an obligation ("the button is named ...", "the deadline is 72 hours") without interpretive hedges; a question requiring the client's decision is put as a question, not as an assumption (§7.13).

15.7 Application and checking

- The rules of §15.2–§15.5 are mandatory from the `RENAR-2` level (§11.5); the forms of §15.6 are a recommendation.
- The check is part of the adversarial review (§7.10.2) and a precondition of the approval of the set version: a violation is recorded as a remark on the wording and removed before approval (§10.7.2). A language violation is by itself not a backward finding — that records a divergence of meaning; the exception is a synonym that produced a terminological conflict (§15.3, item 2).
- A static check of the rules (the operator list, the constructions of rows 2–5 and 8 of §15.4, the length) is recommended as `automation.kind: static` (§9.8.1); it is not a gate of the standard.
- A change of the operator list or of the forms is a change of the standard (§13.9); previously approved set versions are not reworded after the fact.

15.8 Language of writing

An artifact in a language other than English is written in that language in full, except: the identifiers and abbreviations of the standard's closed lists (`BR` , `SR` , `SPEC-*` , `TC` , `ADAPT` , `ACTZ` , `QG-N` , `V1` – `V6`); schema field names (`source.tz-section` , `constrained-by[]`). The markers of the forms of §6.6.3.1 are written in the language of the artifact, in its ordinary orthography. The rule coincides with the policy of the corpus — [reference/06 §1](#).

15.9 Example: before and after

Before:

The system must process requests fast. If something goes wrong, it must correctly inform the user about it and/or log the error. All requests are handled by a manager, who should reply within a reasonable time where possible.

After:

WHEN the user submits a request, the system MUST return an acknowledgement within 2 seconds. IF the processing of a request ends in an error, THEN the system MUST display an error message to the user. IF the processing of a request ends in an error, THEN the system MUST write the error to the errors. Log journal. The manager MUST reply to each request no later than 24 hours after it is received.

One "before" sentence became four statements: three forms of §6.6.3.1 and one business rule of §15.6.2; each has exactly one pair of TCs.

15.10 Links to other chapters

Chapter	Link
01 Scope	The closed list of operators — §1.7.5, row 17
04 Terms	The terms of the standard — §4.2; terminological drift — class 4.11.5
06 Requirements hierarchy	The SR statement forms — §6.6.3.1; the BR body — §6.5.3
07 ADAPT	Term mapping; the terminology category; the form of ACTZ clauses — §7.13
09 Test cases	The TC body sections — §9.4; Pass / Fail — §9.11
10 Lifecycle and QG	The language check as a precondition of version approval, §10.7.2
11 Maturity model	Mandatory from RENAR-2 — §11.5
13 Conformance	A pair of TCs per statement — §13.3.5; changing lists — §13.9
reference/04	The informative lexicon and templates for the AI generator
reference/06	The editorial policy of the standard's text (not of artifacts)