SENAR

Supervised Engineering & Normative AI Regulation

Version 1.3 | 25.03.2026

Authors: Andrey Yumashev, Vadim Soglaev

Table of Contents

SENAR

Supervised Engineering & Normative AI Regulation

Version: 1.3 | **Date:** 25.03.2026 **Authors:** Андрей Юмашев (Andrey Yumashev), Вадим Соглаев (Vadim Soglaev) | **Website:** senar.tech

---

> ***NOTE:*** *The following summary is informative. Normative requirements are defined in Sections 4–13.*

Start Here: Minimum Viable SENAR (MVS)

You are practicing SENAR when these six things are true:

1. **Every AI implementation has a Task** with a goal and acceptance criteria — before work begins.
2. **AI output is verified** by a Supervisor against acceptance criteria — never rubber-stamped.
3. **Tasks cannot start** without goal, acceptance criteria, and requirement link (Context Gate).
4. **Tasks cannot close** unless CI passes, tests pass, and types are clean (Implementation Gate).
5. **Throughput and Lead Time are measured** — you know how many tasks per session and how long they take.
6. **Dead Ends are documented** — when an approach fails, you write down why, so no one repeats it.

That's it. Do these six things and you are practicing the core of SENAR. For a structured entry point, see SENAR Core (8 rules, 2 gates, 2 metrics). Everything else in this standard builds on this foundation.

What Is SENAR

SENAR is a methodology for software development where AI agents are the primary producers of engineering artifacts and humans serve as Supervisors — directing, verifying, and governing AI output.

SENAR addresses a fundamental shift in software engineering: when AI produces the code, the human's value moves from **production** (writing code) to **judgment** (designing context, verifying correctness, making architectural decisions).

Existing methodologies (Scrum, SAFe, Kanban) were designed for teams of humans coordinating with each other. SENAR is designed for Supervisor+AI Pairs producing verified, traceable software.

SENAR Values

1. **Context over Code** — AI output quality is determined by input context quality
2. **Verification over Speed** — correctness is the constraint, not velocity
3. **Knowledge over Experience** — what is not documented does not exist for AI
4. **Enforcement over Agreement** — quality standards enforced through automated gates, not meetings
5. **Judgment over Keystrokes** — human attention on decisions and verification, not on typing

SENAR Core

SENAR Core provides 8 foundational rules for any team. This Standard extends Core with organizational processes, metrics, and governance. Teams adopting SENAR for the first time SHOULD start with SENAR Core. Teams implementing SENAR Core are not required to conform to this Standard but are encouraged to adopt Foundation configuration when ready.

Document Set

Document	Purpose	Audience
SENAR Core	8 foundational rules — entry-level adoption	Any team, individual Supervisors
SENAR Standard (this document)	Normative requirements (SHALL/SHOULD/MAY)	Organizations, auditors
SENAR Guide	Philosophy, interaction patterns, training	Supervisors, adopters
SENAR Reference	Glossary, scaling ratios, economic model, compliance, tooling	Managers, compliance

Normative Language

Per RFC 2119: **SHALL** = required, **SHOULD** = recommended, **MAY** = optional.

Normative requirements appear in Sections 4–13 of this Standard. The Guide and Reference are informative.

Tooling

SENAR is tool-agnostic in choice but tool-dependent in practice. Automated quality gates, metric collection, and knowledge management require tooling support. See SENAR Reference (Tooling Requirements) for detailed capability requirements.

AI Capability Dependence

Some SENAR provisions are capability-dependent: they reference AI behavioral patterns (hallucination types, context window limits, instruction following characteristics) that change with model generations. Organizations SHOULD review capability-dependent provisions when the AI model they use changes substantially (see Section 10.13). The SENAR Guide marks such provisions explicitly.

Empirical Basis and Limitations

SENAR's quantitative guidance (metric baselines, session duration guidelines, cost estimates) is derived from a single reference implementation: 552 tasks, $989 in AI costs, 38 sessions

across 6 microservices. This constitutes a case study, not a controlled experiment. Organizations should treat these numbers as illustrative starting points, not universal targets — which is why Section 9 requires establishing your own baselines before setting targets.

Specific limitations: N=1 organization, self-reported metrics, pre/post design without control group, single AI model family. Independent replication across different organizations, domains, and AI models is needed to validate generalizability.

Intellectual Heritage

SENAR builds on established software engineering foundations: requirements engineering (IEEE 29148), quality cost models (Boehm, 1981), process maturity (CMMI/SEI), flow metrics (DORA, Accelerate), lean manufacturing quality practices (First Pass Yield, Right First Time), and human-AI teaming research. SENAR's contribution is the specific application and codification of these principles for AI-native development — where AI agents are the primary code producers and human engineers serve as Supervisors. The methodology does not claim to invent quality engineering; it claims to adapt it for a production model that did not exist when prior frameworks were created.

Versioning

Changes to normative requirements in the Standard are published as numbered versions (1.0, 1.1, 2.0). Guide and Reference materials may be updated between Standard versions. The changelog is maintained at senar.tech.

1. Scope

1.1 Purpose

SENAR defines a methodology for software development where AI agents are the primary producers of engineering artifacts and human engineers serve as Supervisors — directing, verifying, and governing the AI-driven process.

1.2 Intended Audience

- **Organizations** transitioning to AI-native software development;
- **Supervisors** — engineers who direct AI agents;
- **Managers** responsible for delivery, quality, and cost;
- **Tool vendors** building AI-native development platforms;
- **Auditors** evaluating quality practices of AI-native teams.

1.3 Applicability

This standard applies where:

a) AI agents generate a substantial portion of production artifacts (code, tests, configuration, documentation); b) Human engineers direct, review, and approve AI-generated output; c) Traceability from requirements to delivered artifacts is required; d) Quality assurance is enforced through automated mechanisms.

SENAR is tool-agnostic. It applies to any AI coding platform — autonomous agents, IDE assistants, terminal-based tools, or custom pipelines.

1.4 Out of Scope

a) AI model training, fine-tuning, or evaluation; b) Traditional development where humans write the majority of code; c) Organizational change management; d) Specific tool implementations.

1.5 Relationship to Other Standards

SENAR extends and adapts concepts from established methodologies:

- **SAFe 6.0** — SENAR reimagines SAFe concepts for AI-native teams. SAFe comparison notes are provided throughout the document.
- **ISO 9001:2015** — SENAR Quality Gates support selected ISO 9001 clauses. Organizations should conduct gap analysis for full compliance.
- **ISO/IEC 25010:2023** — SENAR metrics align with the software quality model.
- **Scrum / Kanban** — SENAR borrows iterative delivery and flow measurement while replacing human-centric ceremonies with AI-appropriate alternatives.

SENAR may be extended with domain-specific profiles for regulated industries (medical devices, financial services, aerospace) that add controls required by sector-specific standards.

---

2. Normative References

2.1 Normative References

- **RFC 2119** — Key words for use in RFCs to Indicate Requirement Levels
- **RFC 8174** — Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, May 2017.

2.2 Informative References

These standards are referenced for context and alignment but are not normatively invoked by this Standard.

- **ISO 9001:2015** — Quality management systems — Requirements
- **ISO/IEC 25010:2023** — Systems and software Quality Requirements and Evaluation — Product quality model
- **SAFe 6.0** — Scaled Agile Framework (Scaled Agile, Inc.)
- **Scrum Guide** — The Scrum Guide (Schwaber & Sutherland, 2020)
- **Kanban Method** — Kanban: Successful Evolutionary Change for Your Technology Business (Anderson, 2010)
- **DORA Metrics** — Accelerate: The Science of Lean Software and DevOps (Forsgren, Humble & Kim, 2018)
- **ISO/IEC 12207:2017** — Systems and software engineering — Software life cycle processes
- **IEEE 29148:2018** — ISO/IEC/IEEE 29148:2018, Systems and software engineering — Life cycle processes — Requirements engineering.
- **CMMI** — CMMI Institute, "CMMI for Development, Version 2.0", 2018.

3. Terms and Definitions

For complete glossary, see SENAR Reference.

Configuration notation: Where normative requirements differ by configuration, this standard uses the notation `[Team+: SHALL]` to indicate that a requirement is SHOULD at Team but SHALL at Enterprise configuration. See Section 11 for configuration definitions. For entry-level adoption, see SENAR Core.

3.1 AI Agent

A software system powered by a large language model that generates engineering artifacts under human direction. An AI Agent is provided by an AI Model Provider (3.25) and operates at a specific model version.

AI Agents are not stable, deterministic tools. Model versions differ in capability, hallucination profiles, and instruction following behavior. Model version changes are treated as configuration changes (see Section 10.13 for normative requirements).

3.2 Supervisor

A human engineer who directs AI agents, verifies output, makes architectural decisions, and enforces Quality Gates. Primary mode is AI-directed work, with manual coding as justified exception (Section 4.1).

3.3 Supervisor+AI Pair

The fundamental production unit: one Supervisor working with one or more AI Agents.

3.4 Context

The information provided to an AI Agent to produce correct output: goal, acceptance criteria, constraints, knowledge, and traceability links.

3.5 Task

The atomic unit of tracked work. Has a goal, acceptance criteria, and requirement link.

3.6 Exploration

An investigation without full Task formality. Explorations SHOULD be time-bounded (Section 6.1). If it yields implementation work, a Task is created.

3.7 Session

A time-bounded period of supervised AI work with defined start and end.

3.8 Increment

A scope-bounded batch of work with objectives and planned budget.

3.9 Quality Gate

An automated enforcement point that blocks work progression unless criteria are met.

3.10 Knowledge Entry

A documented decision, pattern, known issue, or dead end stored in a searchable knowledge base.

3.11 Dead End

A documented failed approach with reason for abandonment. A dead end is any investigation that takes more than 15 minutes without producing a usable result. When this threshold is reached, the Supervisor stops, documents the approach and reason for failure, and chooses an alternative path. For normative requirements governing Dead End handling, see Section 10.4.

3.12 Checkpoint

A context preservation action during a Session to prevent work loss.

3.13 Gate Bypass

A documented exception allowing work past a Quality Gate. Requires justification, risk acknowledgment, and remediation plan.

3.14 Federation

Coordination mechanism for multiple Supervisor+AI Pairs across one or more projects: dependency tracking, shared knowledge, cross-project alerts. See Section 5.7 for federation requirements when managing multiple projects.

3.15 Cycle Time

Time from Task start to Task completion (`started_at → completed_at`). Distinguishes execution time from queue time (compare with Lead Time: `created_at → completed_at`).

3.16 Story

An intermediate grouping of Tasks representing a deliverable visible to stakeholders.

3.17 Requirement

A documented, verifiable statement of a need, capability, or constraint that the system must satisfy. SENAR defines three requirement levels: Business Requirement (3.18), System Requirement (3.19), and Task Requirement (3.20). A Task's goal and acceptance criteria constitute requirements at the TR level.

3.18 Business Requirement (BR)

A stakeholder-level need expressed in business terms. Source of all downstream requirements. Typically corresponds to an Increment objective or Epic goal.

3.19 System Requirement (SR)

A system-level capability or constraint derived from one or more Business Requirements. Expressed in terms of system behavior, not implementation. Typically corresponds to a Story

goal.

3.20 Task Requirement (TR)

An implementation-level requirement decomposed from a Business or System Requirement. Corresponds to a Task's goal and acceptance criteria. The lowest level at which requirements are formally managed; test cases are verification artifacts derived from TRs, not a requirement level.

3.21 Requirement Hierarchy

The decomposition chain from business need to implementation unit: BR → SR → TR. Not all levels are required for all work; depth is determined by the Context Architect based on complexity and regulatory context (see Section 8.2).

3.22 Test Model (TM)

A verification artifact derived from Task Requirements that defines how each TR will be verified: test cases, test data, expected results, and verification method (automated test, manual demonstration, measurement). The Test Model is NOT a requirement level — it is the bridge between requirements and verification. In AI-native development, AI typically generates tests from TRs; the Supervisor verifies that generated tests actually exercise the stated acceptance criteria.

The level of Test Model formality scales by configuration — see Section 11 and QG-2 (Section 8.3) for normative requirements.

3.23 Code Documentation

Module-level, API-level, and architectural documentation that serves as persistent context for AI Agents. In AI-native development, code documentation has a dual audience: human Supervisors and AI Agents. Self-contained, machine-readable documentation reduces per-Task context overhead and improves AI output quality.

3.24 Traceability

The ability to trace every engineering artifact back to its originating requirement through a chain of linked references. Bidirectional: every TR traces up to a BR; every BR decomposes to at least one TR. Full traceability chain: BR → SR → TR → TM → Code.

3.25 AI Model Provider

An external service that provides AI model inference capabilities (e.g., cloud AI inference APIs, on-premise model servers). AI Model Providers are de facto suppliers — the AI model is the primary production tool, equivalent to a compiler. Model capabilities, limitations, and pricing change with provider decisions outside the organization's control.

3.26 AI Model Version

A specific release of an AI model, identified by provider designation. Version changes may affect output quality, cost, and behavioral characteristics. Different versions differ in capability, hallucination profiles, cost, and instruction-following behavior. Metric baselines (FPSR, cost/task) are version-dependent — see Section 10.13 for recalibration requirements.

3.27 Scope Creep

Unplanned changes to a Task's implementation that go beyond the stated goal and acceptance criteria. In AI-native development, scope creep manifests when AI agents modify code outside the defined scope boundaries, add unrequested features, or refactor existing code not covered by the Task.

3.28 Hallucination (AI)

AI-generated output that is plausible but factually incorrect: references to non-existent APIs, methods, CLI flags, or packages; fabricated file paths; confidently stated but wrong assertions about system behavior. In dependency contexts, a hallucinated package is one that does not exist in the official package registry or that resolves to an unexpected maintainer.

3.29 WSJF (Weighted Shortest Job First)

A prioritization method calculating the ratio of Cost of Delay to Job Size. Used during Increment Planning (Section 7.1) to order the task pool. Adopted from SAFe without modification.

3.30 Value Stream

An end-to-end flow from stakeholder request to delivered, verified software. At Enterprise configuration, Increments are grouped by value stream with unified budgets (Section 11.3).

3.31 Adversarial Review

Independent review of AI-generated output by an agent that has no access to the generating agent's session context or reasoning. See Section 10.15, L3.

3.32 Agent Dispatch

Delegation of a Task or sub-task to a separate AI agent instance, typically operating in an isolated environment. See Section 5.6.

3.33 Agent Profile

A named configuration of scripts, permissions, and context that defines the capabilities and boundaries of an AI agent performing a specific function. See Section 5.2.

3.34 Structured Tool Protocol

A protocol enabling structured interaction between AI agents and platform services, providing self-describing tool schemas, atomic operations, and audit logging. See Section 5.5.

NOTE: Examples of qualifying protocols include Model Context Protocol (MCP), OpenAI function calling, and custom API interfaces.

3.35 Operational Script

A structured natural-language instruction that defines how an AI agent performs a specific action, containing trigger, preconditions, algorithm, postconditions, and outputs. See Section 5.3.

3.36 Adversarial Detection Rate (ADR)

Metric measuring the density of CRITICAL-severity findings discovered by adversarial review per task that underwent L3 review. Formula: `adversarial_critical_findings / L3_reviewed_tasks`. See Section 9.2.

3.37 Code Standards

A document defining mandatory code quality rules, loaded into AI agent context to guide implementation. Covers security, architecture, database, API, concurrency, and testing patterns. See Section 10.15, L2.

3.38 Latent Defect

AI-generated code that appears correct at surface level — passing automated checks, type validation, and self-review — but contains hidden defects (security bypasses, logic errors, architectural violations) detectable only through independent adversarial review. Latent defects arise when AI generates code by pattern matching rather than semantic understanding. See Section 10.15.

3.39 FPSR (First-Pass Success Rate)

The percentage of Tasks that meet all Acceptance Criteria on the first verification attempt, without requiring rework. See Section 9.1 for definition and measurement requirements.

3.40 Quality Sweep

A structured end-of-Increment review covering metrics, open defects, knowledge base coverage, and traceability completeness. See Section 7.4 for normative requirements.

3.41 Flow Manager

A Supervisor role responsible for cross-team coordination, dependency tracking, and flow efficiency within a Federation. See Section 4.4 for role definition and responsibilities.

3.42 Context Architect

A Supervisor role responsible for designing and maintaining the knowledge architecture, context loading strategies, and documentation standards that enable effective AI-directed work. See Section 4.2 for role definition and responsibilities.

3.43 Knowledge Engineer

A Supervisor role responsible for capturing, structuring, and maintaining the organization's Knowledge Entries, dead ends, and reusable patterns. See Section 4.3 for role definition and responsibilities.

3.44 Verification Engineer

A Supervisor role responsible for designing the Test Model, executing adversarial reviews, and enforcing Quality Gates. See Section 4.5 for role definition and responsibilities.

4. Roles

SENAR defines five responsibility sets. One person MAY cover multiple sets. Responsibilities SHALL be covered, not necessarily by dedicated positions.

For career paths, transition guides, and scaling ratios, see SENAR Guide and Reference.

4.1 Supervisor (SHALL — all configurations)

Directs AI agents, verifies output, makes architectural decisions, enforces Quality Gates.

Responsibilities: a) Define Tasks with goals and acceptance criteria before work; b) Provide structured context to the AI agent; c) Direct AI through execution with course corrections; d) Review all AI output against acceptance criteria; e) Make architectural and trade-off decisions; f) Enforce Quality Gates before closing Tasks; g) Capture knowledge entries during work; h) Maintain Session discipline (checkpoints, duration).

The Supervisor prefers AI generation but MAY write code manually when context preparation cost exceeds manual intervention cost. Manual interventions SHOULD be traceable.

4.2 Context Architect (SHALL — all configurations)

Designs requirements as structured AI input, manages the requirement hierarchy (BR → SR → TR), and ensures traceability. Quality is built at input: a defect in a Business Requirement cascades to all downstream System Requirements, Task Requirements, and ultimately to code. The Context Architect is the primary guardian of input quality.

Foundation (combined with Supervisor — Section 4.8, Section 11.1): At Foundation, the Supervisor absorbs Context Architect responsibilities. Minimum Foundation-level duties: a) Maintain the project context file (e.g., CLAUDE.md) and ensure it reflects current project state; b) Ensure context quality before task start — verify that QG-0 inputs are sufficient and well-structured; c) Design initial task decomposition (BR → TR at minimum).

Team (SHALL — dedicated): a) Maintain the requirement hierarchy (BR → SR → TR) with bidirectional traceability; b) Manage requirement lifecycle: draft → approved → verified → deprecated; c) Ensure requirement quality properties: verifiability (SHALL), consistency, sufficiency, non-redundancy, traceability [Team+: SHALL]; d) Prioritize work using WSJF (Section 3.29); e) Conduct periodic traceability audits to identify orphaned requirements

(requirements without Tasks) and orphaned Tasks (Tasks without requirements); f) Enable requirement reuse by maintaining verified requirements as reusable Knowledge Base entries; g) Design Agent Profiles and manage Operational Scripts (Section 5); h) Review script changes as production process changes (Section 10.14); i) Maintain tool inventory per Agent Profile (Section 5.4).

4.3 Knowledge Engineer (SHALL Team+)

Captures, curates, and maintains the organizational knowledge base.

NOTE: At Foundation configuration, Knowledge Engineer responsibilities are combined with the Verification Engineer role and covered by one person (Section 4.8, Section 11.1). The "Team+" designation means a dedicated role; at Foundation, combined coverage applies. A dedicated Knowledge Engineer is required at Team configuration and above.

Responsibilities: a) Capture Dead Ends, decisions, patterns, known issues; b) Ensure entries are searchable and categorized; c) Review knowledge freshness; deprecate outdated entries; d) Analyze knowledge gaps.

4.4 Flow Manager (SHALL Team+)

Manages Session rhythm, cost tracking, and flow metrics.

NOTE: At Foundation configuration, Flow Manager responsibilities are absorbed by the Supervisor (Section 4.8, Section 11.1). The "Team+" designation means a dedicated role; at Foundation, combined coverage applies.

Responsibilities: a) Monitor Session duration and checkpoint cadence; b) Track cost per Task and Increment against budget; c) Monitor flow metrics (Section 9); d) Facilitate Increment Planning and Retrospective; e) Coordinate dependencies between Pairs (Federation); f) Schedule Quality Sweeps.

At Enterprise scale, expands to coordinate multiple Pairs (Federation Coordinator).

4.5 Verification Engineer (SHALL Team+)

Audits AI-generated output for correctness, security, and architectural conformance.

NOTE: At Foundation configuration, Verification Engineer responsibilities are combined with the Knowledge Engineer role and covered by one person (Section 4.8, Section 11.1). The "Team+" designation means a dedicated role; at Foundation, combined coverage applies.

Responsibilities: a) Conduct Quality Sweeps; b) Audit AI output using the AI Output Review Checklist (see SENAR Guide); c) Review AI-generated tests for coverage and assertion quality; d) Track Defect Escape Rate and identify systemic issues.

4.6 Enterprise Roles (Enterprise configuration only)

At Enterprise scale (10+ Pairs), three additional responsibility sets emerge:

Portfolio Manager: Coordinates multiple Increments across value streams. Manages unified budget, cross-stream dependencies, and strategic prioritization. Ensures AI investment aligns with organizational objectives.

Chief Supervisor: Sets architectural standards across all Pairs. Reviews and approves architectural exceptions. Defines organization-wide AI interaction patterns and context templates. Note: Chief Supervisor defines organization-wide standards; Context Architect (Section 4.2) implements them within specific projects.

Federation Coordinator: Manages cross-project dependency tracking, shared knowledge base governance, and cross-team requirement traceability. Facilitates Federation Sync ceremonies (Section 7.5) at the portfolio level.

These are responsibility sets, not job titles. One person MAY cover multiple Enterprise roles.

4.7 Summary

Responsibility	Team	Enterprise
Supervisor	SHALL (each Pair)	SHALL (each Pair)
Context Architect	SHALL (dedicated)	SHALL (dedicated)
Knowledge Engineer	SHALL (dedicated)	SHALL (dedicated)
Flow Manager	SHALL (dedicated)	SHALL (dedicated)
Verification Engineer	SHALL (dedicated)	SHALL (dedicated)
Portfolio Manager	—	SHALL
Chief Supervisor	—	SHALL
Federation Coordinator	—	SHALL

NOTE: For entry-level adoption (1–2 Pairs), see SENAR Core. The Supervisor role in Core absorbs all responsibilities.

4.8 Role Combinations by Team Size

Team Size	Combination Pattern
1 Pair (Core)	Supervisor covers all responsibilities informally
1–3 Pairs (Foundation)	Supervisor + Context Architect (combined), Knowledge Engineer + Verification Engineer (combined). 2 people cover all roles.
3–5 Pairs (Team)	Context Architect + Flow Manager (combined), Knowledge Engineer (dedicated or shared), Verification Engineer (dedicated), Supervisors (each pair). Minimum 3 distinct role-holders.
5–10 Pairs (Team)	All 5 roles dedicated. Context Architect may still combine with Flow Manager at 5–6 pairs.
10+ Pairs (Enterprise)	All roles dedicated + Enterprise roles (Portfolio Manager, Chief Supervisor, Federation Coordinator).

NOTE: Role combinations are SHOULD-level guidance. Organizations MAY choose different combinations based on team skills and domain. The requirement is that all responsibilities are covered, not that specific combinations are used.

5. Agent Instrumentation

5.1 Overview

AI-native development requires explicit management of the AI agent's behavior, capabilities, and constraints. This section defines how organizations SHALL configure, version, and govern the instruments that control AI agent behavior.

Agent instrumentation operates at three levels:

Level	Artifact	Purpose	Analogy
Behavioral Contract	Project rules file	Define boundaries, prohibitions, conventions	Job description
Operational Scripts	Procedural instructions	Algorithmic steps for specific actions	Machine work instructions
Programmatic Interface	API / Tools / Hooks	Instruments for agent-platform interaction	Machine control panel

Each level SHALL be version-controlled and subject to change management (see Section 10.14).

5.2 Agent Profiles

An Agent Profile defines a specific configuration of scripts, permissions, and context for an AI agent performing a particular function.

Organizations SHALL define at minimum the following profiles:

Profile	Scripts	Access	Purpose
Generator	Implementation, commit, debug	Read/write code and tasks	Primary development
Reviewer	Review, security audit	Read-only code and tasks	Independent verification
Planner	Planning, retrospective	Write epics/stories, read all	Architecture, decomposition
Documenter	Documentation	Write knowledge base	Documentation and knowledge capture
Verifier	Quality sweep, testing	Read all, write findings	Quality audit

Separation of concerns: A Reviewer profile SHALL NOT have write access to the artifacts being reviewed. This separation is the foundation of adversarial review — the same agent cannot both generate and approve its own output.

Organizations MAY define additional profiles. One physical AI agent MAY switch between profiles within a session, provided each profile switch is logged.

NOTE: At Foundation configuration (1–3 Pairs), organizations MAY use fewer than five profiles. The 5-profile requirement is SHOULD at Foundation, SHALL at Team+. Foundation teams typically combine Generator and Reviewer into fewer profiles matching their combined role structure (Section 4.8, Section 11.1).

Team configuration: Organizations SHALL define all five profiles with enforced permission boundaries. Enterprise configuration: Organizations SHALL define all five profiles with enforced permission boundaries and audit trail for profile switches.

5.3 Operational Scripts

An Operational Script is a structured natural-language instruction that defines how an AI agent performs a specific action. Scripts are the primary mechanism for encoding organizational process into agent behavior.

5.3.1 Script Structure

Each script SHOULD contain:

- **Trigger**: When the script is invoked (command, event, condition)
- **Preconditions**: What must be true before execution

- **Algorithm**: Numbered steps with decision points
- **Postconditions**: What must be true after execution
- **Outputs**: What the script produces (artifacts, state changes, reports)

5.3.2 Script Governance

Scripts define agent behavior. Changing a script changes the production process. For the operational rule governing script changes, see Section 10.14.

SHALL (all configurations):

- Operational scripts stored in version control system
- Script changes reviewed before deployment (as code changes)
- Decision to change a script recorded in knowledge base with rationale

SHALL (Team+ configurations):

- Script changes tested on an isolated project before propagation to all projects
- Active script registry maintained with version identifiers
- Rollback capability: any script change can be reverted to previous version
- Script change audit trail maintained

SHOULD:

- Scripts have acceptance criteria (what the script does, does not do, edge cases)
- Script effectiveness tracked via metrics (e.g., FPSR of tasks executed under a given script)

5.4 Programmatic Interface

The Programmatic Interface defines what tools, APIs, and automated actions are available to the AI agent.

5.4.1 Tool Inventory

Organizations SHALL maintain an inventory of tools available to each Agent Profile:

- Which API endpoints are accessible
- Which file system operations are permitted
- Which external services are reachable
- Which actions require human confirmation

5.4.2 Principle of Least Privilege

Each Agent Profile SHALL have access only to the tools required for its function:

- A Reviewer SHALL NOT have write access to production code
- A Generator SHALL NOT have access to deployment credentials
- A Verifier SHALL NOT have access to modify quality gate results

5.4.3 Automated Actions (Hooks)

Organizations MAY define automated actions triggered by events:

- Post-session: metrics collection, knowledge sync
- Post-task-completion: quality gate validation, notification
- Pre-commit: lint check, security scan

Hooks SHALL be version-controlled alongside scripts.

5.5 Security Boundaries

5.5.1 Prompt Injection Defense

Organizations SHALL ensure that AI agents do not process untrusted user-supplied content as instructions. When agents read external data (user input, file contents, API responses), that data SHALL be treated as data, not as commands. Operational Scripts SHOULD include explicit instruction boundaries.

Practical measures:

a) Agent prompts SHALL clearly delimit system instructions from user-supplied data; b) External data ingested by agents (API responses, file contents, database records) SHALL NOT be interpreted as agent commands or modifications to the agent's behavioral contract; c) Organizations SHOULD test agent configurations against known prompt injection patterns as part of Quality Sweeps (Section 7.4).

5.6 Structured Tool Protocol

Organizations SHALL expose platform capabilities through a structured, self-describing tool protocol rather than CLI commands or direct database access. The chosen protocol SHOULD provide:

a) Self-describing schemas — the agent receives tool definitions with parameter types and descriptions, reducing hallucinated arguments; b) Atomic operations — each tool invocation is a single transaction, avoiding the "chained command" failure mode; c) Structured input/output — eliminating parsing errors from text-based CLI output; d) Audit logging — every tool invocation is recorded with parameters and results.

NOTE: Examples of qualifying protocols include Model Context Protocol (MCP), OpenAI function calling, and custom REST/gRPC tool APIs.

CLI as fallback: CLI commands SHOULD remain available when the structured tool protocol is unavailable. Organizations SHALL document which operations have CLI fallback and which are protocol-only.

Direct database access: AI agents SHALL NOT use direct database access for write operations. Read operations MAY use direct access for debugging, but production workflows SHALL use protocol or CLI abstractions.

5.7 Agent Dispatch and Execution Isolation

Agent dispatch — delegating a Task or sub-task to a separate AI agent instance — introduces unique risks:

a) The dispatched agent operates with reduced context (no session history, limited knowledge); b) Multiple dispatched agents may modify the same files concurrently; c) The Supervisor cannot observe the dispatched agent's reasoning in real-time.

Organizations using agent dispatch SHALL:

1. **Isolate**: dispatched agents SHOULD work in isolated working copies to prevent file conflicts; NOTE: Examples of isolation mechanisms include version control worktrees, separate repository checkouts, containerized build environments, and ephemeral cloud workspaces.
2. **Scope**: dispatch prompts SHALL include explicit boundaries — which files to modify, which to leave unchanged;
3. **Review**: all dispatched agent output SHALL undergo L3 Adversarial Review (Section 10.15) — agent dispatch is the highest-risk scenario for latent defects. For Foundation configuration, L2 Review with High-tier checklist MAY substitute when independent agent access is unavailable (Section 10.15);
4. **Context**: dispatch prompts SHOULD include relevant Code Standards, architectural constraints, and known issues from the knowledge base;
5. **Limit**: organizations SHALL define a maximum parallel dispatch count per Supervisor (Section 10.7 applies).

Dispatched agents SHALL NOT:

- Modify files outside the scoped boundary without explicit approval;
- Commit directly to shared branches;
- Access other agents' isolated working copies or session state.

Pattern: Supervisor dispatches → agent works in isolated copy → agent returns result → Supervisor reviews → Supervisor merges to main branch.

5.8 Federation — Scaling SENAR Across Projects

When an organization manages multiple projects (a "federation"), SENAR practices scale with additional coordination requirements.

5.8.1 Project Independence

Each project in a federation SHALL maintain its own:

- Task tracker and session history
- Knowledge base (patterns, known issues, dead ends)
- Metric baselines and targets
- Agent configuration (profiles, scripts, permissions)

5.8.2 Cross-Project Coordination

A federation SHOULD designate a coordination project (analogous to a SAFe Release Train Engineer) responsible for:

- Cross-project dependency tracking
- Federated knowledge routing (a pattern discovered in Project A that affects Project B)
- Aggregate metrics (federation-level throughput, cost, quality)
- Shared Code Standards and review criteria

5.8.3 Knowledge Routing

Knowledge entries SHALL be scoped:

- **Project-specific**: patterns, known issues, and decisions relevant only to one project — stored in that project's knowledge base;
- **Cross-project**: patterns affecting multiple projects (e.g., API contract changes, shared library updates) — stored in the coordination project and routed to affected projects;

- **Global**: methodology-level insights — stored in the coordination project, available to all.

Organizations SHALL define routing rules: which knowledge types are automatically shared vs. manually promoted.

Cross-project knowledge entries SHALL require approval from the receiving project's Supervisor before entering that project's active context. Global knowledge entries SHALL require coordination project Supervisor approval. Knowledge entries that reference security-sensitive topics (authentication, authorization, encryption, secrets, CORS, CSRF, permissions) SHALL be flagged for human review regardless of routing rules.

5.8.4 Federated Metrics

Federation-level metrics aggregate project metrics:

Metric	Federation Computation
Throughput	Sum of project throughputs (tasks/session across all projects)
FPSR	Weighted average by project task count
DER	Weighted average by project task count
ADR	Weighted average by project agent task count
Cost	Sum of project costs

Organizations SHALL NOT compare raw metric values across projects with different stacks, team sizes, or maturity levels. Comparison SHOULD use normalized metrics (e.g., cost per story point by complexity).

5.9 Portability

This standard is not bound to any specific AI agent, tool, or platform.

Operational Scripts SHOULD be written in structured natural language interpretable by any AI agent of sufficient capability:

- Clear algorithmic steps (not platform-specific SDK calls)
- Explicit inputs and outputs per step
- Conditional logic expressed in natural language
- No assumptions about specific tool implementations

The Programmatic Interface MAY be platform-specific (structured tool protocols, function calling APIs, etc.), but the Behavioral Contract and Operational Scripts SHALL be portable

across AI agent implementations.

When migrating between AI platforms, organizations SHALL:

- Verify that all Operational Scripts execute correctly on the new platform
- Recalibrate baseline metrics (Section 10.13: AI Model Governance)
- Update the Programmatic Interface layer while preserving Contract and Script layers

---

6. Units of Work

6.1 Exploration

Time-bounded investigation without full Task formality.

a) Explorations SHOULD be time-bounded (organization defines limit); b) If Exploration yields implementation, a Task SHALL be created before changes are committed; c) Explorations SHOULD be logged with a summary.

6.2 Task

The atomic unit of tracked work.

Required Attributes

Attribute	Description
Goal	What must be accomplished
Acceptance Criteria	Verifiable conditions defining done
Requirement Link	Link to parent Story, Business Requirement, System Requirement, or Task Requirement (SHALL)
Work Type	Functional category (development, architecture, QA, documentation)

Lifecycle States

Transition	Gate
planning → active	QG-0
active → done	QG-2
done → merged (Team+)	QG-3
merged → released (Team+)	QG-4
active → blocked	None
blocked → active	None

At Team configuration and above, the `done` state leads to merge review (QG-3) and release approval (QG-4).

6.3 Story

An intermediate grouping of Tasks representing a deliverable visible to stakeholders. Links Tasks to requirements.

Required Attributes

Attribute	Description
Title	One-sentence description of the deliverable
Acceptance Criteria	Story-level verifiable conditions defining done (independent of individual Task AC)
Linked Tasks	References to all Tasks that compose this Story
Requirement Link	Link to parent Business Requirement (BR) or System Requirement (SR) (SHALL)

Completion Criteria

A Story is complete when all its Tasks have reached the `done` state AND the story-level Acceptance Criteria have been verified by the Supervisor.

6.4 Session

A time-bounded period of supervised AI work.

a) Organizations SHALL establish and document a maximum Session duration; b) Checkpoints SHALL be performed at intervals documented by the organization; c) Sessions SHALL begin with context loading and end with metrics capture and handoff; d) Ceremonies MAY be automated tooling commands; e) Version control discipline: commits SHALL be atomic; secrets detection SHALL be automated; AI changes SHALL be reviewed for scope creep.

6.5 Increment

A scope-bounded batch of work with defined objectives, planned budget, and risk register.

a) Each Increment SHALL have 3–5 measurable objectives; b) Task pool SHOULD be prioritized (WSJF recommended); c) Each Increment ends with Quality Sweep and Retrospective.

7. Ceremonies

Ceremonies handle human strategic decisions. Quality enforcement is handled by Gates (Section 8).

7.1 Increment Planning (SHALL)

Define objectives, task pool, budget, and risks for the upcoming Increment.

Participants	Context Architect (leads), Flow Manager, Supervisors
Frequency	Once per Increment

Outputs: objectives, prioritized tasks, planned budget, risk register.

7.2 Session Start (SHALL)

Load context, select tasks, verify environment.

Participants	Supervisor
Overhead	Minimal (MAY be automated)

7.3 Session End (SHALL)

Capture metrics, write handoff, record knowledge.

Participants	Supervisor
Overhead	Minimal (MAY be automated)

SHALL outputs: summary, task states, handoff (next steps, warnings), metrics, dead ends (if any). **SHOULD outputs:** knowledge entries for decisions/patterns/known issues.

7.4 Quality Sweep (SHALL)

Periodic comprehensive audit of codebase and knowledge base.

Participants	Verification Engineer (leads)
Frequency	At cadence documented by the organization

Scope: security, code quality, test health, configuration drift, knowledge freshness, architectural conformance, dependency health, AI-specific issues (duplication, scope creep accumulation, TODO debris).

7.5 Federation Sync (SHOULD; SHALL at Team+)

Coordinate multiple Supervisor+AI Pairs.

Participants	Flow Manager (leads), Supervisors
Frequency	Regular cadence (daily or every 2–3 days recommended)

Agenda: status per Pair, dependency updates, shared component changes, integration issues.

7.6 Delivery Review (SHOULD)

Demonstrate working software to stakeholders, collect feedback, obtain acceptance.

Participants	Context Architect (leads), Supervisor, Stakeholder
Frequency	Per deliverable milestone

7.7 Increment Retrospective (SHALL)

Quantitative review: planned vs actual cost, throughput trend, FPSR, DER, ADR, knowledge capture rate.

Participants	Flow Manager (leads)
Frequency	End of each Increment

Output: specific, measurable, time-bounded, assigned improvement actions.

Organizations SHOULD allocate time for innovation, learning, and AI tooling experimentation.

7.8 Summary

Ceremony	Mandate	Frequency
Increment Planning	SHALL	Per Increment
Session Start	SHALL	Per Session
Session End	SHALL	Per Session
Quality Sweep	SHALL	Periodic (org-defined)
Federation Sync	SHALL (Team+)	Regular cadence
Delivery Review	SHOULD	Per milestone
Increment Retrospective	SHALL	Per Increment

8. Quality Gates

Automated enforcement points. Implemented as code, not checklists. See SENAR Guide for AI Output Review Checklist.

8.1 QG-0: Context Gate (Task start)

Quality is built at input. QG-0 ensures that a Task has sufficient, well-structured context before AI-directed work begins.

SHALL criteria: goal defined, acceptance criteria verifiable, requirement or story linked (SHALL), work type assigned.

SHALL criteria (Team+): acceptance criteria are independently verifiable (each can be tested or measured separately); at least one negative scenario (error case, invalid input, or boundary condition) is included in acceptance criteria.

SHALL criteria (all configurations, security-relevant tasks): security surface declared — tasks touching auth, user input, data storage, payment, or external APIs SHALL identify the threat surface and include at least one security-related acceptance criterion. See SENAR Core (Start Gate, criterion 5) and Section 8.7.

SHOULD: plan, complexity estimate, relevant knowledge identified.

NOTE: Foundation configuration inherits Core's Start Gate requirements (scope boundaries, negative scenario) even though they are listed as "Team+" SHALL criteria above. At Foundation, QG-0 = Core Start Gate checks + Standard QG-0 structure. The "Team+" criteria (independently verifiable AC, negative scenario) are already practiced at Core level; Foundation formalizes them as gate criteria.

8.2 QG-1: Requirements Gate (Story/Increment level)

Ensures requirements are defined, approved, decomposed, and of sufficient quality before implementation begins.

SHALL criteria [Team+]: Business Requirement (BR) exists and is approved; decomposition to Task Requirements (TR) is complete at appropriate depth; all TRs have verifiable acceptance criteria; no orphaned requirements (requirements without implementation Tasks).

SHOULD criteria (Team+: SHALL): requirements satisfy consistency (no contradictions between TRs within a Story); requirements satisfy sufficiency (normal, boundary, and error scenarios covered); non-functional requirements specified where applicable (performance, security, accessibility).

Decomposition Depth

The Context Architect SHALL determine decomposition depth [Team+] based on complexity and regulatory context:

Context	Required Depth	Example
Standard feature, Team config	BR → SR → TR (3 levels)	BR: "Support OAuth" → SR: "OAuth provider flow" → TR: Task AC
Complex/regulated, Enterprise config	BR → SR → TR → TM (formal Test Model)	Full traceability chain for audit compliance

NOTE: For simple features (BR → TR, 2 levels), see SENAR Core.

Requirement Quality Properties

Requirements submitted to QG-1 SHALL be verifiable (each can be tested, measured, or demonstrated). Requirements SHOULD (Team+: SHALL) satisfy:

a) **Consistency** — no contradictions with existing requirements at the same or higher level; b) **Sufficiency** — the set of TRs covers the parent requirement (SR or BR) completely; c) **Non-redundancy** — no duplicate requirements across Stories; d) **Traceability** — every TR traces up to a BR; every BR decomposes to at least one TR.

Requirement Change Management

NOTE: Requirement Change Management applies at Team+ configuration.

When an approved BR or SR changes after implementation has begun:

a) The change SHALL be documented with rationale and the identity of the person who authorized it; b) Impact analysis SHALL identify all downstream requirements and Tasks affected by the change; c) Affected Tasks that are already `done` SHALL be flagged for re-verification — their QG-2 approval is invalidated; d) The changed requirement SHALL pass QG-1 again before new implementation begins; e) Supervisors of affected Tasks SHALL be notified of the change.

Scaling: at Team, impact analysis SHALL be supported by tooling (requirement parent/child links). At Enterprise, requirement changes SHALL be version-controlled and subject to change review (see Section 11.3, Requirements-as-Code).

8.3 QG-2: Implementation Gate (Task done)

SHALL criteria: CI passes, tests pass, static analysis passes (including type checking where applicable), no new lint violations, acceptance criteria verified by Supervisor, no security vulnerabilities detected by scanning tools.

SHOULD: coverage meets threshold, no duplication, docs updated, knowledge entry if decision/dead end. Team+: generated tests reviewed against Test Model (TM) — AI-generated tests SHALL exercise the stated acceptance criteria, not just achieve coverage.

SHOULD (Team+: SHALL): AI-modified dependency manifests (the project's declared dependency files — e.g., package.json, requirements.txt, go.mod, Cargo.toml) reviewed for hallucinated packages — packages that do not exist in the official registry or that resolve to unexpected maintainers, typosquatted packages (near-name-matches of legitimate packages), and dependency confusion (internal vs. public namespace collisions).

8.4 QG-3: Verification Gate (Merge — Team+ configs)

SHALL criteria: acceptance tests pass, security scan clean, no regressions, code reviewed (risk-based — see 8.7).

SHOULD: performance within SLA, accessibility compliance, cross-service integration verified.

AI Output Review Minimum Criteria

At QG-3, the following AI-specific checks SHALL be performed:

a) Generated code does not call non-existent APIs, methods, or CLI flags (hallucination check); b) All imported packages exist in the project's dependency manifest (dependency validation); c) Generated code follows project architectural patterns and conventions (pattern conformance); d) Generated tests exercise the stated acceptance criteria, not just achieve coverage (test validity); e) No hardcoded credentials, API keys, or secrets in generated code (secrets check).

These criteria replace dependency on the informative AI Output Review Checklist in the SENAR Guide with normative minimum requirements.

8.5 QG-4: Acceptance Gate (Release)

SHALL criteria: Delivery Review conducted OR stakeholder approval recorded, QG-3 still passing, staging environment (production-equivalent deployment target) verified, stakeholder acceptance recorded.

8.6 Enforcement Rules

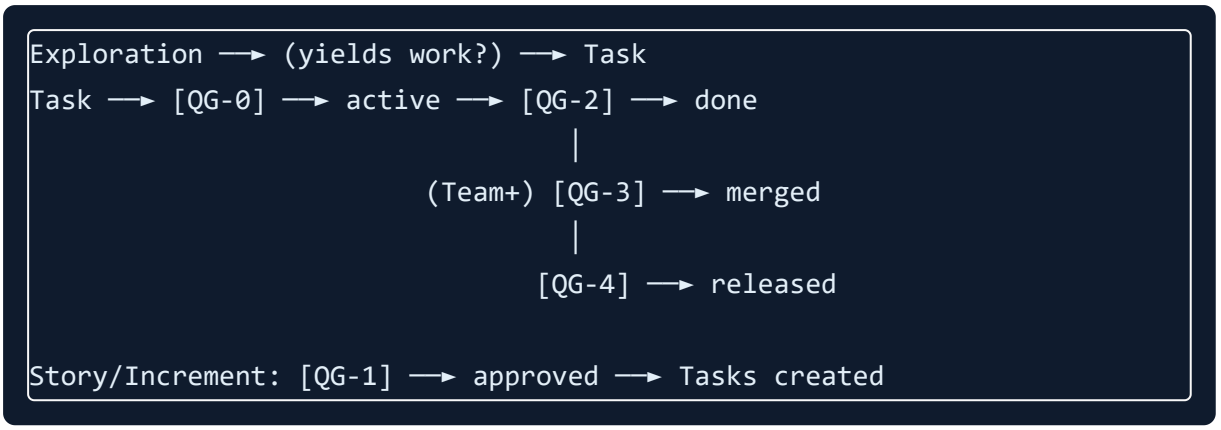
a) Gates SHALL be automated wherever feasible. Human judgment criteria require explicit recorded approval. b) Every gate execution SHALL produce an audit record. c) Gates SHALL NOT be bypassed without documented Gate Bypass (justification + risk + remediation + senior approval). d) Organizations SHOULD track Gate Bypass rate.

8.7 Risk-Based Review

Risk Level	Examples	Review
High	Security, auth, payment, data migration, architecture	SHALL: peer review + security review (all configurations)
Standard	Feature, UI, business logic	QG-2 automated + Supervisor verification statement
Low	Docs, config, trivial fixes	QG-2 automated sufficient

Security review for high-risk changes: For changes affecting authentication, payment processing, personal data handling, or cryptographic operations, security review SHALL be performed at ALL configuration levels, not only Enterprise. This is a mandatory (SHALL) requirement regardless of team size. Organizations developing security-sensitive systems SHALL adopt security controls (QG-3 with AI Output Review Minimum Criteria, risk-based code review) regardless of configuration level.

8.8 Gate Pipeline



8.9 Summary

Gate	Applied At	Core	Foundation	Team	Enterprise
QG-0	Task start	YES (Start Gate)	YES	YES	YES
QG-1	Story/Increment	—	—	YES	YES
QG-2	Task done	YES (Done Gate)	YES	YES	YES
QG-3	Merge	—	—	YES	YES
QG-4	Release	—	—	YES	YES

NOTE: SENAR Core defines QG-0 (Context Gate) and QG-2 (Implementation Gate) as its two quality gates. Foundation configuration inherits both gates from Core (Section 11.1).

8.10 Security Requirements Cross-Reference

One consolidated view of all security-related requirements across the Standard. Use this table when auditing security coverage or onboarding a new configuration.

Requirement	Where Defined	Configuration
AI Output Review Minimum Criteria (secrets check, hallucination check)	QG-3 (8.4)	Team+
High-risk change review — human approval required (auth, payment, data, crypto)	8.7 + Section 10.15 L3	All configs
Security surface declaration at task start	QG-0 (8.1)	All configs
Auth coverage verification	AI Output Review Checklist item 15 (SENAR Core)	High tier
Input validation	AI Output Review Checklist item 6 (SENAR Core)	Standard tier
Secrets detection in sessions	Section 6.4	All configs
Agent dispatch review	Section 5.6 + Section 10.15	Team+

NOTE: "All configs" means the requirement applies regardless of team size — including Foundation teams. See 8.7 for the normative statement on security review applicability.

---

9. Metrics

Organizations SHALL establish baselines by measuring for at least 3 Increments before setting targets. Targets are organization-specific, not universal.

9.1 Mandatory (SHALL)

#	Metric	Formula	Measures
1	**Throughput**	tasks_done / sessions	Delivery performance
2	**Lead Time**	completed_at - created_at	Delivery speed
3	**First-Pass Success Rate**	single_cycle_tasks (completed without rework) / total × 100%	Context quality
4	**Defect Escape Rate**	post_done_defects / done × 100%	Gate effectiveness

A post-done defect is a defect task explicitly linked to a completed parent task that introduced the defect. The link SHALL be recorded in the task tracker.

9.2 Recommended (SHOULD; SHALL in Team+)

#	Metric	Formula	Measures
5	**Knowledge Capture Rate**	entries / tasks	Organizational memory
6	**Cost Predictability**	actual_cost / planned_cost × 100%	Estimation accuracy
7	**Cost per Task**	total_cost / tasks (by complexity)	Efficiency
8	**Manual Intervention Rate**	manual_tasks / total × 100%	AI-first adherence
9	**Cycle Time**	completed_at - started_at	Execution speed (vs Lead Time which includes queue time)
10	**Adversarial Detection Rate**	adversarial_critical_findings / L3_reviewed_tasks	Latent defect density

Manual Intervention Rate requires Supervisor self-reporting (flag on task indicating manual code was written). Organizations SHOULD define what constitutes "manual intervention" (e.g., any hand-written production code, or only tasks done entirely without AI).

Adversarial Detection Rate (ADR) measures the density of CRITICAL-severity findings discovered by adversarial review (Section 10.15, L3) per task that underwent L3 review. Target: organizations SHOULD aim for ADR < 0.5 (fewer than one CRITICAL finding per two reviewed tasks). An ADR of 0 indicates either excellent AI output quality or insufficient review rigor — organizations SHOULD distinguish between the two.

For ADR computation, "L3_reviewed_tasks" means tasks that underwent L3 Adversarial Review. Tasks that did not receive L3 review (e.g., Low-risk tasks where L3 was skipped per Section 10.15) are excluded from the denominator. This ensures ADR reflects review effectiveness, not review coverage.

Knowledge Capture Rate target calibration: organizations with established knowledge bases SHOULD set KCR targets that reflect diminishing returns. A target of 1.0 (one entry per task) is appropriate for greenfield projects. For mature projects (>500 tasks), a target of 0.33 (one entry per three tasks) better reflects the natural rate of novel knowledge discovery. Organizations SHALL document their KCR target rationale.

9.3 Collection

Metrics 1–5 and 9 SHALL be collected automatically from task tracking and session data. Metrics 6–7 require cost tracking integration. Metric 8 requires Supervisor self-reporting. Cost Predictability requires Flow Manager assessment at Increment Retrospective.

Metric 10 (ADR) SHALL be collected from adversarial review findings recorded during L3 review (Section 10.15). Organizations SHALL maintain a record of review findings per task, classified by severity, to compute ADR.

Metric computation SHALL include all tasks without epoch-based filtering or exclusion of historical data. Organizations SHALL NOT exclude tasks from metric computation based on creation date, migration status, or tooling version. Rationale: epoch filters introduce complexity, create maintenance burden, and mask data quality issues. Historical data naturally dilutes as new data accumulates, converging metrics to their true values over time. If early data is known to be unreliable (e.g., pre-automation manual entries), this SHALL be documented as a known limitation, not filtered.

Cost Predictability (metric 6) requires organizations to record planned cost before implementation begins. In practice, planned cost estimation for AI-assisted tasks is unreliable — AI execution time is non-deterministic and model pricing varies. At Team configuration, Cost Predictability SHALL be tracked but baselines may be provisional during the first 3 Increments. At Enterprise configuration, Cost Predictability SHALL be tracked with established baselines. Cost per Task (metric 7) provides a more actionable proxy for cost management in early adoption.

Multi-agent and multi-session configurations: when multiple AI agents work in parallel (multiple concurrent sessions), per-session metrics (throughput, session duration, cost per session) reflect individual agent performance, not aggregate team output. Organizations using multi-agent configurations SHOULD additionally track aggregate metrics at the Increment level. Token and cost attribution in multi-agent scenarios SHOULD be recorded per-agent, with aggregate totals available for Increment-level reporting.

---

10. Operational Rules

10.1 Task Before Implementation (SHALL)

Implementation SHALL NOT begin without a Task. Explorations are exempt; if they yield code, a Task SHALL be created before commit.

10.2 Session Duration (SHALL)

Organizations SHALL establish, document, and enforce a maximum Session duration. Basis SHOULD be documented. The documented maximum SHOULD be informed by empirical session data. As a starting guideline, sessions exceeding 180 minutes show diminishing returns in most AI-assisted workflows.

10.3 Checkpoint Cadence (SHALL)

Checkpoints SHALL be performed at intervals not exceeding a duration documented by the organization. The documented interval SHOULD be no greater than the organization's maximum acceptable context loss in the event of a session interruption.

10.4 Dead End Documentation (SHALL)

Failed approaches SHALL be documented as Dead End knowledge entries (approach, reason, alternative). A dead end is any investigation that takes more than 15 minutes without producing a usable result. When this threshold is reached, the Supervisor SHALL stop, document the approach and reason for failure, and choose an alternative path.

10.5 Periodic Audit (SHALL)

Quality Sweeps SHALL be conducted at a cadence documented by the organization, reviewed at each Retrospective. The documented cadence SHOULD be no less frequent than once per 3 Increments. For Foundation configuration, the recommended cadence is monthly (Section 11.1), which satisfies this requirement when Increments are 2 weeks or shorter.

10.6 Version Control (SHALL)

a) Commits SHALL be atomic; b) Secrets detection SHALL be automated; c) AI changes SHALL be reviewed for scope creep.

10.7 Parallel Agent Limit (SHALL)

Supervisors SHALL NOT exceed the organization's defined parallel agent limit. A common starting limit is 3 concurrent agents per Supervisor; organizations SHOULD calibrate based on task complexity and Supervisor capacity.

10.8 Complexity-Cost Calibration (SHALL)

Organizations SHALL maintain cost baselines per task complexity level.

10.9 Knowledge Capture (SHALL)

Organizations SHALL establish and track a Knowledge Capture Rate target.

10.10 Requirement Traceability (SHALL Team+)

Organizations SHALL maintain traceability from Business Requirements through System Requirements to Task Requirements. Every Task SHALL link to at least one requirement or Story. Requirements SHALL be stored in a versioned, searchable system.

10.11 Code Documentation as Context (SHALL)

Code documentation (module-level documentation (docstrings, doc comments, or equivalent), API descriptions, architecture docs) is not a post-hoc artifact — it is active context for AI Agents. Well-documented code reduces the context volume a Supervisor must provide per Task and enables AI to produce correct output with less explicit instruction.

Organizations SHALL maintain code documentation that includes: (a) module/package purpose statement, (b) public API contracts with parameter and return types documented, (c) architectural boundary description identifying dependencies and integration points. The documentation SHALL be machine-readable (structured comments, API description formats, or equivalent). Documentation SHOULD be updated as part of the same Task that changes the code (QG-2 criterion: "docs updated if API changed").

In AI-native development, documentation has a dual audience: human Supervisors and AI Agents. Documentation that is useful only to humans (e.g., "see John for details") is useless for AI. Documentation SHALL be self-contained and machine-readable.

10.12 Context Hygiene (SHALL)

Supervisors SHALL NOT include production credentials, API keys, personally identifiable information (PII), or regulated data (financial, health) in AI Agent context unless the AI model provider has a current data processing agreement and the data classification permits it. Organizations SHALL maintain a context classification policy defining which data categories are prohibited in AI context. Placeholder or synthetic data SHOULD be used when AI assistance is needed for tasks involving sensitive domains.

10.13 AI Model Governance (SHALL)

AI model providers are external suppliers. The AI model is the primary production tool — equivalent to a compiler. Organizations SHALL:

a) Record the AI model identifier and version used for each Session; b) Recalibrate metric baselines (FPSR, cost per task, throughput) when the active model version changes substantially — a change in model generation constitutes a substantial change; c) At Team+: evaluate new model versions before adoption — compare FPSR and cost per task on a representative task set; d) At Enterprise: maintain a formal model approval process — new models SHALL be tested against acceptance criteria for the organization's domain before production use; e) Document known model-specific limitations and hallucination patterns as Knowledge Entries.

NOTE: SENAR provisions that reference AI behavior (hallucination detection heuristics in Guide, session duration recommendations, parallel agent limits) are capability-dependent. Organizations SHOULD review these provisions when model generations change substantially.

10.14 Script Change Management (SHALL)

Organizations SHALL treat operational script changes as production process changes:

- Version-controlled (Section 10.6)
- Reviewed before deployment
- Decision recorded in knowledge base (Section 10.9 applies)

Team+ configurations SHALL additionally:

- Test script changes on isolated environment before propagation
- Maintain version registry of active scripts per project
- Ensure rollback capability for any script change

For the full definition of Operational Scripts and their governance, see Section 5.3.

10.15 AI Output Quality Verification (SHALL)

AI-generated code SHALL undergo quality verification before commit. Three verification levels are defined:

Level	Method	Configs
L1: Automated	Static analysis — file size, function size, complexity metrics, security pattern checks, lint	All (SHALL)
L2: Verification Statement	The person or agent who did the work writes a structured verification statement confirming what was checked against acceptance criteria and Code Standards	All (SHALL)
L3: Adversarial Review	An independent agent (different model or context-free) reviews without access to the generating agent's reasoning	SHALL (Team+)

L3 Adversarial Review requirements: a) The reviewing agent SHALL NOT have access to the generating agent's session context or reasoning; b) At least one reviewer SHOULD be a "cold" reviewer — an agent with zero prior context of the task; c) Findings SHALL be classified by severity (CRITICAL, HIGH, MEDIUM) and recorded; d) CRITICAL findings SHALL block commit until resolved; e) Organizations SHALL track Adversarial Detection Rate (ADR) — see Section 9.

L3 review SHALL be applied based on risk level (see Section 8.7):

- **High risk** (security, auth, payment, data migration, architecture): For High risk changes (Section 8.7), L3 review SHALL be applied regardless of configuration level; at least one reviewer SHALL be a human Supervisor, not an AI agent; security review SHALL be performed (see Section 8.7);
- **Standard risk** (feature, UI, business logic): L3 SHALL be applied (Team+);
- **Low risk** (docs, config, trivial fixes): L3 MAY be skipped with documented justification.

NOTE: L2 Verification Statement addresses quality defects (logic errors, style violations, acceptance criteria coverage) but does NOT constitute a security control. For security assurance, L3 is the minimum effective verification level. The L2 verification statement is a structured record confirming what was checked — it is not a review by an independent party.

The person or agent who did the work writes the statement, confirming each acceptance criterion was verified and how.

NOTE: Latent defects in AI-generated code are characteristically difficult to detect because they satisfy automated quality checks while containing subtle logical, security, or architectural flaws. Common patterns include: returning True without validation, null equality comparison bypasses, trusting HTTP headers without verification, empty configuration values that silently disable security checks, and unreachable "safety" code that masks incomplete control flow understanding.

Agent dispatch (delegating implementation to a sub-agent) is the highest-risk scenario for latent defects, as the dispatched agent operates with reduced context. Organizations using agent dispatch SHALL apply L3 review to all dispatched agent output.

10.16 Summary

#	Rule	Team	Enterprise
1	Task Before Implementation	SHALL	SHALL
2	Session Duration	SHALL	SHALL
3	Checkpoint Cadence	SHALL	SHALL
4	Dead End Documentation (>15 min threshold)	SHALL	SHALL
5	Periodic Audit	SHALL	SHALL
6	Version Control	SHALL	SHALL
7	Parallel Agent Limit	SHALL	SHALL
8	Complexity-Cost Calibration	SHALL	SHALL
9	Knowledge Capture	SHALL	SHALL
10	Requirement Traceability	SHALL	SHALL
11	Code Documentation as Context	SHALL	SHALL
12	Context Hygiene	SHALL	SHALL
13	AI Model Governance	SHALL	SHALL
14	Script Change Management	SHALL	SHALL
15	AI Output Quality Verification	SHALL	SHALL

NOTE: For entry-level adoption, see SENAR Core which defines 8 foundational rules.

11. Configurations

NOTE: For individual adoption, see SENAR Core — 8 rules, 2 quality gates, 2 metrics. This Standard defines three organizational configurations: Foundation, Team, and Enterprise.

11.1 SENAR Foundation (1–3 Pairs)

The bridge between Core and Team. For small teams that need more structure than Core but aren't ready for full Team process.

Category	Requirements
Base	All SENAR Core rules apply
Responsibilities	3 combined responsibility sets covered by 2 people (Section 4.8): Supervisor + Context Architect (combined), Knowledge Engineer + Verification Engineer (combined). Flow Manager responsibilities are absorbed by the Supervisor.
Ceremonies	3: Session Start, Session End, Quality Sweep (monthly). NOTE: Foundation conducts Increments (Section 6.5) but omits formal Increment Planning and Retrospective ceremonies. Planning is implicit in Session Start; retrospective insights are captured in Quality Sweep.
Quality Gates	2: QG-0 (Context Gate) + QG-2 (Implementation Gate)
Metrics	4: Throughput, Lead Time, FPSR, Defect Escape Rate
Rules	Core 8 + Rule 2 (Standard 10.2, Session Duration), Rule 4 (Standard 10.4, Dead End Documentation), Rule 9 (Standard 10.9, Knowledge Capture) = 11 rules
Knowledge Base	Shared, accessible to all team members and AI sessions

> ***Note:*** *At Foundation, session duration monitoring (Rule 10.2) is the Supervisor's responsibility, since Flow Manager is not a separate role at this configuration (Section 4.8).*

> ***Note:*** *Core rules and Standard rules use different numbering. Core Rule 8 (Capture Knowledge) expands into Standard Rules 10.4 (Dead End Documentation) and 10.9 (Knowledge Capture). Foundation adds these as explicit, tracked requirements alongside Standard Rule 10.2 (Session Duration).*

> ***Note:*** *Section 6.4 (secrets detection in sessions) implicitly requires elements of Rule 10.6 (Version Control). Organizations adopting Foundation SHOULD also adopt Rule 10.6 to ensure atomic commits, automated secrets detection, and AI scope-creep review.*

Adoption Path (week by week)

- **Week 1 — Core habits + Session Start/End.** All team members read SENAR Core and begin applying its 8 rules daily. Introduce Session Start (context load + goal declaration) and Session End (output summary + knowledge entry) as team ceremonies. Assign the 3 combined responsibilities — roles may overlap.
- **Week 2 — Dead End documentation + Knowledge capture.** Activate Rule 4 (Dead End Documentation): every blocked path gets a knowledge entry before work pivots. Activate Rule 9 (Knowledge Capture): every significant decision and non-obvious finding goes into the shared KB. After two weeks the team has a working knowledge base, not an empty one.
- **Week 3 — Quality Sweep (first monthly).** Run the first Quality Sweep: review the session log, check which dead ends were documented, identify recurring friction points. This is a retrospective with output — concrete adjustments to thresholds, session duration, or task decomposition depth.
- **Week 4 — Metrics baseline.** Start measuring all 4 metrics: Throughput (tasks completed per week), Lead Time (task start to done), FPSR (first-pass success rate at QG-2), Defect Escape Rate (bugs found after done). Record baseline values — no targets yet, just calibration. Adjust thresholds to your domain.

Foundation vs Core — What's Different

Core is individual discipline: one Supervisor, two quality gates, two metrics, all enforced at the personal level. Foundation adds **team coordination layer**:

- Shared knowledge base visible to all team members and AI sessions (not just the author's local context)
- Session Start/End as **team ceremonies** — synchronization points, not just individual habits
- Monthly Quality Sweep — collective review of process health, not individual retrospection
- 2 additional metrics (FPSR + DER) that only make sense when multiple people share a definition of "done"

The cognitive overhead is low. Foundation is designed to be adopted incrementally on top of Core habits already in place.

FAQ for Foundation Teams

When do we move to Team? When any of the following applies: the team reaches 3+ Pairs and coordination overhead is visible; you need cross-pair dependency tracking or knowledge federation; you want QG-1 (Requirements Gate) or QG-3 (Verification Gate); you want dedicated (non-overlapping) responsibilities.

Can we skip Foundation and go directly to Team? Yes. If the team already has process discipline — structured task decomposition, code review culture, retrospectives — start at Team. Foundation is a bridge, not a mandatory stop.

What if we're exactly 3 pairs? Evaluate experience level. If everyone is experienced with structured AI-assisted development, go Team — the overhead is manageable. If the team has mixed experience or is just starting with AI workflows, stay Foundation for 4–8 weeks to build habits before adding federation and all 5 gates.

11.2 SENAR Team (3–10 Pairs)

Category	Requirements
Responsibilities	All 5 dedicated (SHALL)
Ceremonies	All 7 (per Section 7)
Quality Gates	All 5: QG-0 through QG-4 (SHALL)
Metrics	All 10 (SHALL)
Rules	All 15 (SHALL)
Federation	Cross-Pair dependency tracking, shared KB (SHALL)
Risk-Based Review	QG-3 differentiated by risk (Section 8.7)
Security Review	SHALL for high-risk changes (auth, payment, data, crypto) — see Section 8.7

11.3 SENAR Enterprise (10+ Pairs)

All Team requirements, PLUS:

Category	Additional
Portfolio Management	Increments grouped by value stream with unified budget
Additional Responsibilities	Portfolio Manager, Chief Supervisor, Federation Coordinator
Compliance	QG audit trails for ISO/regulatory requirements
Governance	Gate Bypass review, architectural exceptions, budget oversight
Requirements-as-Code	Requirements stored in VCS, CI validates traceability, change review required (SHALL)

For scaling ratios and enterprise guidance, see SENAR Reference.

NOTE: Enterprise configuration describes target state for large-scale adoption. Organizations at this scale should validate ratios and adapt coordination mechanisms to their context.

11.4 Comparison

Element	Core	Foundation	Team	Enterprise
Pairs	1	1–3	3–10	10+
Rules	8	11	15	15
Quality Gates	2	2 (QG-0, QG-2)	5 (QG-0..QG-4)	5 + compliance
Metrics	2	4	10	10 + portfolio
Responsibilities	1 (Supervisor)	3 (combined)	5 (dedicated)	5 + portfolio
Ceremonies	0	3	7	7 + portfolio
Knowledge Base	Recommended	Required	Required + federation	Required + federation
Security Review	Checklist	Checklist	SHALL (high-risk)	SHALL + formal audit
Tooling	None	Recommended	Required	Required

11.5 Migration

```
SENAR Core ──► Foundation ──► Team ──► Enterprise
 + 3 roles (combined)  + dedicated roles    + portfolio management
 + 3 ceremonies        + federation         + value streams
 + 2 more metrics      + all gates          + compliance
 + session mgmt        + all metrics/rules  + governance
```

Each step is incremental. Pilot with a subset of Pairs first. Typical timelines:

- Core → Foundation: 2-4 weeks after team has internalized Core habits
- Foundation → Team: 2-3 months, when team reaches 3+ Pairs or needs federation
- Team → Enterprise: when organization has 10+ Pairs and needs portfolio governance

---

12. Maturity Model

12.1 Levels

Level 1: Ad Hoc

AI used opportunistically. No tasks, no gates, no metrics, no documented knowledge.

Level 2: Supervised (aligns with SENAR Core)

Tasks before work. AI output verified. QG-0 and QG-2 enforced. 2 metrics tracked (FPSR + Throughput). Dead Ends documented (>15 min threshold). Sessions have start/end with handoffs.

Key indicator: Every AI implementation has a Task; output is verified.

Level 3: Measured (aligns with Team)

All 5 responsibility sets covered. All 5 gates automated. All 10 metrics with baselines and targets (including ADR tracked per task). Knowledge capture structured. Federation active. Architectural Decision Records (ADRs) tracked. Data-driven decisions.

Key indicator: Baselines established; metrics reviewed at every Retrospective.

> ***Note:*** *Levels 4 and 5 are aspirational targets based on industry patterns (CMMI, DORA). No SENAR implementation has been independently validated at these levels. They are provided as directional guidance for organizations seeking continuous improvement.*

Level 4: Managed (aspirational)

Cost predictability consistent. FPSR improving. Defect Escape Rate below threshold. Knowledge base actively improves AI context quality. Process improvements are experimental and measured.

Level 5: Optimizing (aspirational)

Organization defines own patterns. Value streams optimized. Process experiments routine. Knowledge base is competitive advantage.

NOTE: Levels 4–5 are aspirational. See the note above Level 4 for details.

12.2 Assessment Dimensions

Organizations MAY be at different levels across dimensions:

Dimension	L1	L2	L3
Task Discipline	None	Tasks before work	Full lifecycle + traceability
Quality Gates	None	QG-0 + QG-2	All 5 automated
Metrics	None	2 (Core)	4 mandatory + 6 recommended
Knowledge	Undocumented	Dead Ends captured	Structured + target rate
Verification	None	Supervisor self-verifies	Independent verification, adversarial review and ADR tracking
Cost	Unknown	Tracked	Baselined and predicted
Requirements	Undocumented	Task goal + AC	Full hierarchy (BR→SR→TR) + quality properties

12.3 Progression

Organizations SHOULD progress sequentially. Each level builds on the previous.
Organizations MAY focus on weakest dimensions first.

---

13. Conformance

13.1 Claiming Conformance

An organization claiming conformance to SENAR SHALL:

a) Identify the applicable configuration: Foundation, Team, or Enterprise (Section 11); b) Implement all SHALL requirements of the applicable configuration; c) Document any SHOULD requirements that are not implemented, with rationale; d) Maintain evidence of implementation (audit records, metric data, task tracker records).

A conformance claim SHALL use the following format: "[Organization] conforms to SENAR v[version] [Configuration] configuration, [self-declared | peer-assessed | independently audited], as of [date]."

13.2 Conformance Levels

Level	Description	Evidence
Self-declared	Organization assesses itself against SENAR requirements	Internal assessment record
Peer-assessed	Assessment conducted by a SENAR practitioner from another organization	Peer assessment report
Independently audited	Assessment by a qualified auditor with SE process assessment experience	Formal audit report

Organizations MAY claim any conformance level. Higher levels provide stronger evidence for stakeholders, clients, and regulators.

13.3 SENAR Core

Teams implementing SENAR Core are not required to conform to this Standard but are encouraged to adopt Foundation configuration when ready. SENAR Core conformance is declared separately per the SENAR Core document.

13.4 Partial Conformance

Organizations MAY claim partial conformance by specifying which sections they conform to (e.g., "SENAR Team conformant for Sections 6–9"). Partial conformance claims SHALL explicitly list the included sections.

Partial conformance claims SHALL include at minimum Sections 6 (Units of Work), 8 (Quality Gates), 9 (Metrics), and 10 (Operational Rules) — these constitute the irreducible core of SENAR practice. A claim covering only definitional sections (Sections 1–4) is not a valid partial conformance claim.

13.5 Non-Conformance Handling

When a SHALL requirement cannot be met, the organization SHALL document the non-conformance with: justification, risk acknowledgment, remediation plan, and approval by a role senior to the requestor. Non-conformance records are distinct from operational Gate Bypasses (Section 3.13) — they apply at the process level, not the task level.

13.6 Maintaining Conformance

Organizations claiming conformance SHOULD reassess at each Increment Retrospective (Section 7.7). Conformance claims SHOULD be re-evaluated after: major process changes, AI model generation changes (Section 10.13), or organizational scaling across configuration tiers.

SENAR Quick Start: 5 Minutes to Better AI Development

You use AI to write code. SENAR makes the result reliable.

No meetings. No mandatory certifications. These habits are the **SENAR Core** — 8 rules, 2 quality gates, 2 metrics — everything you need to start. Adoption overhead: under 1 hour, about 5 minutes per session after that.

The 6 Habits

BEFORE you tell the AI what to build

1. Write the WHAT and the DONE

Before starting, write two things:

- **Goal** — what must be accomplished (one sentence)
- **Acceptance Criteria** — how you'll know it's done (numbered list, each independently testable)

Bad: "Add login functionality" Good: "Implement email/password login. AC: 1. POST /auth/login returns JWT on valid credentials. 2. Returns 401 on invalid password. 3. Returns 422 on missing email field. 4. Token expires in 24h."

Why this matters: AI output quality equals input quality. A vague goal produces plausible-looking code that fails in production. A precise goal with clear acceptance criteria produces testable, correct code on the first try.

2. Set boundaries

Tell the AI what NOT to touch:

- "Change ONLY the users/ directory"
- "Do NOT modify the database schema"
- "Follow patterns from auth/router.py"

Without boundaries, AI will confidently refactor half your codebase to "improve" things you didn't ask about.

WHILE the AI works

3. Verify against your criteria, not your intuition

Don't just glance at the code and think "looks right." Check each acceptance criterion:

- AC 1: Does POST /auth/login return JWT? → Check the test or run it.
- AC 2: Does it return 401 on invalid password? → Check the test.
- AC 3: Missing email → 422? → Check the test.

If there's no test for a criterion, the criterion isn't verified.

4. Document dead ends

When an approach fails, write one sentence about why:

- "Tried bcrypt for password hashing — import fails on Python 3.14, switched to argon2"
- "SQLAlchemy async session: can't use lazy loading, need selectinload"

This takes 10 seconds and saves hours — for you next week, for your teammate, for any AI that reads your knowledge base.

AFTER it's done

5. Run the tests

If tests pass AND all acceptance criteria are met → done. If not → not done. No exceptions, no "it probably works."

6. Capture what you learned

If you discovered something non-obvious during this task, write it down:

- A decision you made and why
- A pattern that worked well
- A gotcha that surprised you

> ***Note — 6 habits vs. 8 Core rules:*** *These 6 habits cover the essence of the 8 SENAR Core rules. The full Core adds two explicit practices: a tiered verification checklist for latent defects (Rule 5) and root-cause analysis before patching symptoms (Rule 7). If you are working in a regulated or high-stakes context, read the full Core document before relying on this Quick Start alone.*

Before & After: Real Data

These numbers come from one project (6 microservices, 552 tasks, $989 in AI costs, 38 sessions). They are illustrative, not universal — your numbers will differ. SENAR requires you to establish your own baselines before setting targets.

Without structured process (sessions 1–3, ad hoc):

- Tasks started without acceptance criteria → AI produced code that "looked right" but failed edge cases
- No dead ends documented → same failed approaches repeated across sessions
- No session discipline → 200+ minute marathon sessions with noticeable efficiency decline
- Defects discovered after "done" → estimated rework cost ~$105 per escaped defect (estimated from project data: average rework cost of defects found after task completion)

With SENAR habits adopted (sessions 4–38):

- Every task has goal + AC before start → First-Pass Success Rate improved to 85%+ (tasks correct on first try)
- Dead ends documented → repeated failures eliminated for documented cases
- Sessions capped at 120 min with checkpoints → consistent throughput, no context crashes
- QG-0 blocks taskless work → zero "what was this for?" tasks

Caveat: This is a single team's experience (N=1), not a controlled study. The improvement conflates methodology adoption with natural team learning. Independent replications are needed — and that's why we published the standard, so others can measure too.

The cost of SENAR: ~5 minutes session overhead + 1–3 minutes per task. **What you avoid:** rework from vague requirements, repeated dead-end approaches, marathon sessions with declining output.

That's It

These habits correspond directly to the **SENAR Core** rules — the self-contained subset of the SENAR Standard. Everything else in the methodology builds on this foundation.

What you get:

- Fewer "works on my machine" surprises
- AI produces correct code on the first try more often (we measure this — it's called First-Pass Success Rate)
- Knowledge accumulates instead of evaporating between sessions
- You can hand off to another developer (or your future self) without losing context

What it costs: ~5 minutes session overhead + 1–3 minutes per task for goals, criteria, and verification.

Next Steps

You want to...	Read
Read the formal SENAR Core document (8 rules, 2 gates, 2 metrics)	**SENAR Core**
See a complete task start-to-finish	Guide: Worked Example
Set up SENAR with your AI tool	Guide: Tool Integration (Claude Code, Cursor, Copilot)
Adopt SENAR in an existing codebase	Guide: Legacy Adoption
Learn about requirement levels	Guide: Requirements Engineering
Understand the philosophy	Guide: Philosophy
Upgrade from Core to the full Standard	Guide: Transition — Core to Standard
Scale to a team	Standard, Section 11: Configurations
Evaluate your current practice	Standard, Section 12: Maturity Model

---

SENAR Guide: Philosophy

This document expands on the SENAR Values and explains the six pillars that underpin the methodology.

SENAR Values

1. **Context over Code** — AI output quality is determined by input context quality. Invest in requirements, not in coding speed.
2. **Verification over Speed** — AI generates at machine speed. Correctness, not velocity, is the constraint.
3. **Knowledge over Experience** — AI has no memory between sessions. What's not documented doesn't exist for AI.
4. **Enforcement over Agreement** — Quality gates as automated code, not meetings people can skip.
5. **Judgment over Keystrokes** — Human attention on decisions (what to build, is it correct), not on typing code.

Pillar 1: Context-First (Quality at Input)

The quality of AI output is a direct function of input context quality.

The cascade principle: Quality is built at input, not checked at output. A defect in a Business Requirement propagates to all downstream System Requirements, Task Requirements, and ultimately to code. By the time a defective requirement reaches AI execution, it produces plausible-looking code that passes automated checks but solves the wrong problem. No amount of testing at the output (QG-2, QG-3) can catch a requirement that was wrong from the start.

This is why SENAR invests in requirement quality (QG-0, QG-1) before implementation begins — not as bureaucracy, but as the highest-leverage quality investment.

Requirements ARE context. A well-defined Business Requirement that decomposes into clear System Requirements and Task Requirements IS the primary input to AI-generated code quality. The requirement hierarchy (BR → SR → TR) is not paperwork — it is the structured context that determines whether AI produces correct output.

Context components:

- Task goal and acceptance criteria (= Task Requirements)
- Requirement links to parent Story/BR/SR (= traceability)
- Architectural constraints and conventions
- Relevant knowledge (decisions, dead ends, gotchas)
- Examples and anti-patterns
- Scope boundaries (what NOT to change)

Common failures:

- *Vibe prompting* — vague instructions without structure. Appropriate for trivial tasks; dangerous for complex work.
- *Context dumping* — flooding AI with unstructured information. Critical constraints get buried.
- *Implicit assumption* — expecting AI to know conventions without documentation.
- *Ambiguous requirements* — requirements that a human would clarify in conversation but AI interprets literally or hallucinates a resolution. Unlike human teams, AI does not ask "did you mean X or Y?" — it picks one silently.

Pillar 2: AI-First, Not AI-Only

The Supervisor's primary mode is AI-directed work. Manual coding is a justified exception, not a prohibition.

When manual intervention is appropriate:

- Micro-fixes (1–3 lines) cheaper than context preparation
- AI stuck in a loop on wrong approach
- Infrastructure config where AI hallucinates
- Time-critical hotfixes
- AI got 90% right — easier to fix 3 lines than re-explain

Common failures:

- *Shadow coding* — writing code and hiding it from traceability
- *Micromanaging* — rewriting >30% of output instead of improving context

- *Rubber stamping* — accepting without verification

Pillar 3: Enforcement over Ceremony

Quality is enforced through automated gates, not meetings.

AI agents don't attend meetings, feel accountability, or learn from retrospectives. The only reliable quality mechanism is automated enforcement.

Purpose	Mechanism
Decide what to build	Ceremony
Verify code quality	Gate
Review with stakeholder	Ceremony
Verify requirement met	Gate

Pillar 4: Knowledge Persistence

What is not documented does not exist for AI.

Code documentation is context, not afterthought. In AI-native development, code documentation serves a dual purpose: it helps human Supervisors understand the system AND reduces the context volume AI needs per Task. A well-documented module (docstring explaining purpose, public API contracts, architectural boundaries) means the Supervisor doesn't need to re-explain these things in every Task goal. AI reads the docs, understands the module's role, and produces code that fits.

Undocumented code forces the Supervisor to compensate: longer Task goals, more explicit constraints, more scope boundaries. This is expensive context that should live in the code itself. Documentation that says "handles OAuth flow for Google and GitHub, stores tokens encrypted in session table, refreshes automatically" saves 5 lines of Task context on every auth-related task.

Rule 9.11 (Code Documentation as Context) makes this a SHALL requirement: code documentation sufficient for AI to understand module purpose, API contracts, and boundaries without reading the full implementation.

What to capture: decisions (with rationale), patterns, gotchas, dead ends (highest reuse value), observations.

Knowledge delivery to AI: static context files, search APIs, MCP integrations, or RAG. Choose deliberately.

Knowledge lifecycle: `current` → `needs_review` (flagged as potentially stale) → `deprecated`. Stale entries actively harm output quality — they give AI incorrect context.

Pillar 5: Interaction Patterns

AI supervision is a dialogue, not a one-shot command.

Pattern	When to Use
Plan-then-Execute	Complex tasks: ask AI for plan → review → approve → execute
Iterative Refinement	Most tasks: generate → review → course correct → improve
Example-Driven	UI/patterns: "do it like this but with X change"
Negative Example	Known pitfalls: "don't do X because Y"
Checkpoint-and-Verify	Multi-step: AI does step 1 → verify → step 2
Constraint Fence	Scope control: "change ONLY files X, Y. Do NOT touch A, B"
Rollback-and-Retry	Wrong direction: git rollback, restart with different context
Exploration	Unknown territory: investigate before committing to approach

Context Window Management

- AI context degrades as conversation grows (saturation)
- Progressive disclosure: provide information as needed, not all at once
- Strategic checkpoints clear context and restart fresh
- For large codebases: targeted file reads over bulk dumps
- Handoffs serialize essential context for new sessions
- If AI starts confusing things, it's not the model — the context is full

Hallucination Management

Common types: non-existent APIs/methods/CLI flags, non-existent files, correct-looking code with subtle edge case bugs, confident but wrong assertions.

Detection heuristics:

- Excessive confidence in novel solutions (red flag)
- "Suspiciously perfect" code handling every edge case
- References to paths/APIs you don't recognize
- Always run the code — don't trust AI's claim about what it does

---

Pillar 6: Empirical Calibration

Every rule and target should be calibrated to your data.

SENAR provides formulas, not targets. Organizations establish baselines by measuring for 3+ Increments, then set targets based on their own reality.

Common failures:

- *Cargo culting* — adopting another org's targets
- *Premature optimization* — aggressive targets before understanding baselines
- *Metric fixation* — optimizing for metric rather than outcome

---

SENAR Guide: AI Output Review Checklist

> ***NOTE — Relationship to SENAR Core***
>
> *SENAR Core includes an updated **Verification Checklist** (26 items, 3 tiers: **Standard / High / Critical**) that supersedes this checklist for Core adopters.*
>
> ***Tier mapping:** Tier 1 (this guide) ≈ Standard (Core) | Tier 2 ≈ High | Tier 3 ≈ Critical*
>
> *This Guide checklist is retained for teams adopting SENAR through the Standard (team-level entry point) who have not yet moved to Core. If your team uses SENAR Core, use the Core Verification Checklist instead.*

When reviewing AI-generated output, check for these AI-specific issues.

The Checklist

#	Check	What to Look For
1	**Scope**	Did AI modify files outside the task scope? (most common AI error)
2	**Deletions**	Did AI silently remove or replace existing working code?
3	**Phantom imports**	Are all imported packages in the dependency file?
4	**Dependency versions**	Are specified versions real and published?
5	**Hardcoded values**	Magic numbers, URLs, credentials, API keys in code?
6	**Over-engineering**	Unnecessary abstractions, patterns, or generalization?
7	**Duplication**	New code that duplicates existing utilities?
8	**Test quality**	Do tests verify behavior, or just mirror implementation?
9	**Test tampering**	Did AI modify tests to pass instead of fixing the code?
10	**Security**	Open CORS, hardcoded tokens, SQL without parameterization?
11	**Edge cases**	Happy path works — what about null, empty, boundary, concurrent?
12	**Naming**	Does AI follow project naming conventions?
13	**Commit scope**	Is the commit atomic and focused, or a kitchen sink?
14	**Null guard before comparison**	`None == None` is `True` in Python (JS: `null === null` is true; most languages have equivalent null-equality traps) — access checks bypassed when both sides null?
15	**Empty config bypass**	Security check skipped when config value is empty string? (`if secret and ...` fails open)
16	**Header trust**	X-Forwarded-For, X-Partner-ID, Content-Length used for security without proxy validation?
17	**IDOR**	Resource accessed by ID without verifying user's access to that resource? Auth ≠ authorization.

18	**Return True shortcut**	Access control function returns True/grants access without explicit ownership validation?
19	**Format string injection**	`str.format(**untrusted_dict)` — Python format supports attribute access, enables injection (JS: template literals with `eval()`; C/C++: `printf` format strings; any language: string interpolation with untrusted input)
20	**God functions/files**	Functions >50 lines, files >400 lines — strongest signal of unrefactored AI output
21	**Unreachable safety code**	`return False` after exhaustive exception handling — AI added "just in case" but can never execute
22	**Swallowed exceptions**	`catch/except` blocks that discard errors (`except Exception: pass` , `catch(e) {}` , or logging at debug level) — hides real failures. Check for null/None/nil returns that silently mask error conditions
23	**Unsafe deserialization**	pickle.loads, yaml.load without SafeLoader, eval/exec on untrusted input, JSON prototype pollution? (Java: `ObjectInputStream`; JS: `eval(JSON)`; any language: deserializing untrusted data without validation)

Priority Tiers

Tier 1 — Always check (every task): Items 1–3 (scope, deletions, phantom imports) and 8–9 (test quality, test tampering). These catch the most frequent and most dangerous AI defects. A 5-item check takes under 2 minutes.

Tier 2 — Security-sensitive tasks (auth, payment, data, API): Items 10 (security), 14–18 (null guard, empty config, header trust, IDOR, return True). These catch latent defect patterns — AI output that looks correct but fails under adversarial conditions. Identified through adversarial audits of production AI-generated code.

Tier 3 — Deep review (complex tasks, agent dispatch output): All remaining items (4–7, 11–13, 19–23). Apply when reviewing complex features, refactors, or any output from dispatched agents (Rule 15 L3).

Usage

Print this list. Use it at QG-2 (Implementation Gate) for every Task. Over time, some checks become automatic habits — but keep the list visible for new Supervisors.

SENAR Guide: End-to-End Walkthrough

Adoption Paths

SENAR has two adoption paths. Start with Core; upgrade to Standard when your needs outgrow it.

Core Path (Individual or Small Team)

1. **Adopt the 8 rules** — Task Before Code, Scope Boundaries, Verify Against Criteria, Tests Verify Requirements, Check for Latent Defects, Zero Tolerance for Incomplete Work, Fix Causes Not Symptoms, Capture Knowledge.
2. **Enforce the 2 gates** — Start Gate (goal + AC + negative scenario + scope before implementation) and Done Gate (all AC verified, checklist passed, knowledge captured).
3. **Measure the 2 metrics** — FPSR (First-Pass Success Rate) and DER (Dead End Rate). Establish baselines for 3+ cycles before setting targets.

That is all. No roles, no ceremonies, no session management. The walkthrough below demonstrates this path.

Standard Path (Team or Organization)

1. **Start with Core** — adopt all 8 rules, both gates, both metrics.
2. **Add Foundation configuration** — 1–3 Pairs; add session discipline (Session Start/End, monthly Quality Sweep); Supervisor combines Context Architect, Knowledge Engineer, and Verification Engineer roles. 11 rules, 4 metrics (adds Throughput, Lead Time), QG-0 + QG-2.
3. **Add Team configuration** — introduce dedicated roles (Context Architect, Flow Manager, Verification Engineer), full ceremonies (Increment Planning, Quality Sweep, Retrospective).
4. **Add Enterprise configuration** — federation coordination, cross-project metrics, portfolio governance, compliance audit trails.

The Team Walkthrough section below demonstrates the Standard path at Team configuration.

Core Walkthrough (Solo Supervisor)

One person, one AI agent, one session. Building a REST API endpoint.

Session Start (2 min): Open terminal. Load previous handoff: "Auth module done. Next: user profile endpoint." Select task `impl-user-profile`.

Context Gate check: Goal: "GET /users/me returns current user profile." AC: "1. Returns 200 with user data. 2. Returns 401 without token. 3. Includes company info." Story link: `user-management`. → Passes.

Execution: Direct AI with Constraint Fence:

> *"Implement GET /users/me. AC: [above]. Follow patterns from auth/router.py. Change ONLY users/ directory."*

(This example uses Python/FastAPI. The same pattern applies to Java/Spring Boot @RequestMapping, Go/Gin router.GET, TypeScript/NestJS @Get().)

AI generates router + test. Review with checklist: scope ✓, imports ✓, no hardcoded values ✓, tests check behavior not implementation ✓. One issue: AI didn't test the 401 case.

> *"Add test for unauthenticated request → 401."*

AI adds test. Re-review: clean.

Implementation Gate: CI green, 6/6 tests pass, mypy clean, AC verified. → Done. (For other stacks: Java uses javac + SpotBugs, Go uses go vet + staticcheck, TypeScript uses tsc --strict.)

Dead End capture: None this task — straightforward.

Session End (3 min): 4 tasks done, 75 minutes. Handoff: "Profile, settings, avatar upload done. Next: password change. Warning: avatar upload needs file size validation."

Total overhead: 5 minutes (start + end). No ceremonies, no roles, no meetings. Just discipline.

> ***The section below demonstrates Team-level SENAR.*** *If you're using Core only, you can stop here — the Core walkthrough above is your complete workflow.*

Team Walkthrough (3 Pairs)

A complete SENAR cycle for Team configuration. 3 Supervisor+AI Pairs, web application.

Increment Planning

Context Architect leads:

1. Reviews backlog: 32 candidate Tasks across 6 Stories
2. Applies WSJF: Authentication scores highest (blocks other work, moderate size)
3. Assigns: Pair A → auth (6 tasks), Pair B → orders CRUD (8 tasks), Pair C → CI/CD (5 tasks)
4. Risks: "Auth library choice may need exploration" → creates Exploration
5. Budget: 19 tasks selected for this Increment

A Supervisor's Session

Session Start (2 min): Load handoff → "Auth model done. Next: login endpoint." Select task `impl-login-endpoint`. Dev environment green.

QG-0 (automatic): Goal ✓, AC ✓ (4 criteria), requirement link ✓, work type ✓ → passes.

Execution (Plan-then-Execute + Constraint Fence):

> *"Implement POST /auth/login. AC: [criteria]. Follow patterns in auth/models.py. Use PyJWT. Config in settings.py. Change ONLY auth/ directory."*

(This example uses Python/FastAPI with PyJWT. The same constraint fence pattern applies to any stack: Java/Spring Security with jjwt, Go with golang-jwt, TypeScript/NestJS with @nestjs/jwt.)

AI generates router, service, tests. Supervisor reviews with checklist:

- Scope: ✓ only auth/ files
- Phantom imports: ⚠ AI imported `python-jose` but project uses `PyJWT`
- Hardcoded values: ✓ token secret from env
- Edge cases: ⚠ no expired token test

Iterative Refinement:

> *"Two issues: 1) Use PyJWT not python-jose. 2) Add expired token test."*

AI corrects. Re-review: clean.

QG-2 (automatic): CI ✓, 14/14 tests ✓, mypy ✓, lint ✓, AC verified ✓, security ✓ → passes.

Knowledge capture: "Gotcha: AI defaults to python-jose for JWT even when PyJWT is in requirements. Always specify library in context."

3 tasks done, 110 minutes.

Session End (5 min): Metrics saved, handoff written, 1 knowledge entry.

Quality Sweep

Quality Sweep procedure:

1. Select 3-5 recently completed tasks (random or risk-weighted)
2. For each task, verify: (a) AC have evidence, (b) checklist was applied at correct tier, (c) knowledge was captured
3. Record findings as observations in the knowledge base
4. If FPSR drops below target, discuss root causes in the next Team Sync

After 6 sessions, Verification Engineer audits:

Finding	Action
Duplicate validation utility (exists in shared/)	Fix task: refactor
3 TODO comments left by AI	Fix task: resolve
Zero test coverage for order deletion	Fix task: add tests
Inconsistent error format between modules	Fix task: standardize

Knowledge: "Pattern: AI undertests delete operations — add to standard AC template."

Increment Retrospective

Metric	Value
Throughput	7.3 tasks/session
Lead Time (median)	35 minutes
FPSR	74%
DER	5.3%
KCR	0.42
Cost Predictability	110%
MIR	12%

Actions: improve context template (JWT library), add "test deletion" to AC template, review QG-2 for concurrency tests.

SENAR Guide: Transition from Traditional Roles & Core to Standard

Upgrading from SENAR Core to SENAR Standard

When to Upgrade

SENAR Core is sufficient for individual developers and small teams (1-2 people). Consider upgrading to the full Standard when:

- **Team grows beyond 3 people** — you need roles (Context Architect, Flow Manager, Verification Engineer) to coordinate work across Supervisors.
- **Multiple projects** — cross-project dependencies require federation coordination, not ad-hoc communication.
- **Compliance or regulatory needs** — you need audit trails, formal requirement traceability, and documented quality evidence.
- **Knowledge silos appear** — tribal knowledge is not making it into the knowledge base, and different Supervisors produce inconsistent results.
- **DER plateaus** — Dead End Rate stops improving because knowledge is captured but not systematically reused across the team.

What Standard Adds Over Core

Dimension	SENAR Core	SENAR Standard
Rules	8 rules	15 rules (adds session management, knowledge lifecycle, code documentation, federation)
Gates	Start Gate + Done Gate	5 gates: QG-0 (Context) + QG-1 (Requirement) + QG-2 (Implementation) + QG-3 (Verification) + QG-4 (Deployment)
Metrics	FPSR + DER	10 metrics (adds Throughput, Lead Time, KCR, MIR, Cost per Task, Cost Predictability, and more)
Roles	Supervisor (implicit)	Supervisor, Context Architect, Flow Manager, Verification Engineer, Knowledge Engineer
Ceremonies	None	Session Start/End, Increment Planning, Quality Sweep, Increment Retrospective
Configurations	Not applicable	Foundation, Team, Enterprise — progressive adoption levels (Core serves as the entry point)
Session management	Not prescribed	Duration limits, checkpoint cadence, handoff discipline
Multi-team	Not covered	Federation Sync, cross-project dependencies, portfolio metrics
Maturity model	Not applicable	6 dimensions, 5 maturity levels per dimension

Migration Steps

If you are already using SENAR Core, migration to Standard is incremental — nothing is unlearned.

Step 1: Map your Core practice to Standard (Day 1) Your 8 Core rules map directly to Standard rules. Your Start Gate maps to QG-0. Your Done Gate maps to QG-2. FPSR and DER remain your primary metrics. SENAR Core is the entry point to the Standard — you are ready for Foundation configuration.

Step 2: Add session discipline — Foundation configuration (Week 1) Introduce formal Session Start (load context, select tasks, verify environment) and Session End (write handoff, capture metrics, document knowledge). Cap sessions at 4–6 hours. Add a monthly Quality Sweep. This is Foundation: 11 rules, 3 combined roles (Supervisor covers Context Architect +

Knowledge Engineer + Verification Engineer), 4 metrics (add Throughput and Lead Time). Suitable for 1–3 Pairs.

Step 3: Add Team ceremonies (Week 2-4) When you have 3+ Supervisors:

- **Increment Planning** — Context Architect leads: review backlog, apply WSJF prioritization, assign tasks to Pairs.
- **Quality Sweep** — Verification Engineer audits: sample recent work for architectural consistency, test quality, knowledge freshness.
- **Increment Retrospective** — Flow Manager leads: review metrics, identify process improvements, set targets.

Step 4: Add organizational metrics (Month 2+) Beyond FPSR and DER, begin tracking: Throughput (tasks/session), Lead Time, Knowledge Capture Rate (KCR), Cost per Task, Cost Predictability. Establish baselines for 3 Increments before setting targets.

Step 5: Add governance (when needed) Enterprise configuration (10+ Pairs): federation coordination across projects, QG-3 (independent verification), QG-4 (deployment gate), requirement traceability, audit trails.

Core Rules to Standard Rules Mapping

Core Rule	Standard Equivalent
1. Task Before Code	Rule 1 (S10.1) + QG-0 (S8.1)
2. Scope Boundaries	QG-0 SHOULD criteria + Guide habit
3. Verify Against Criteria	QG-2 (S8.3) + Rule 15 L2 (S10.15)
4. Tests Verify Requirements	QG-2 test criteria + QG-3 test validity (S8.4)
5. Check for Latent Defects	Rule 15 (S10.15) + AI Review Checklist (Guide)
6. Zero Tolerance for Incomplete Work	QG-2 enforcement (S8.6)
7. Fix Causes, Not Symptoms	Rule 5 Periodic Audit (S10.5) + Guide failure modes
8. Capture Knowledge	Rule 4 Dead End Documentation (S10.4) + Rule 9 Knowledge Capture (S10.9)

Role Mapping

Current Role	SENAR Responsibility	Transferable Skills	Skills to Develop
Senior Developer	Supervisor	Code review, architecture, domain knowledge, debugging	Context design, AI interaction, delegation over execution
Tech Lead	Senior Supervisor	Architecture, trade-offs, mentoring	AI-first mindset, reduced hands-on coding
QA Engineer	Verification Engineer	Verification, edge case thinking, acceptance criteria	AI output patterns, hallucination detection
DevOps Engineer	Supervisor (infra)	Automation, CI/CD, infrastructure	AI-directed config, context for infrastructure
Product Owner	Context Architect	Requirements, prioritization, stakeholder management	Structuring requirements for AI consumption, traceability
Scrum Master	Flow Manager	Facilitation, process optimization	Metrics-driven management, cost tracking

Supervisor Career Path

Level	Focus	Key Competency
Junior Supervisor	Single tasks, close verification	Following patterns, thorough review
Supervisor	Multiple tasks, architectural decisions	Context design, trade-off judgment
Senior Supervisor	Cross-project impact, mentoring	Architectural direction, reviewing others
Staff Supervisor	Process optimization, methodology	QG design, maturity assessment

Common Transition Challenges

"I'm faster typing it myself" — True for trivial tasks. For moderate+, context design + AI generation is faster than manual coding AND produces tests, documentation, and traceable output. Measure it.

"I don't trust AI output" — Good instinct. That's what QG-2 and the Review Checklist are for. Verification is your core skill now, not trust.

"My value was writing code" — Your value was solving problems. You still solve problems — but now through direction and judgment, not keystrokes. The problems get bigger.

SENAR Guide: SAFe Comparison

SENAR is a standalone methodology inspired by SAFe concepts, designed for AI-native teams. Organizations using SAFe for human-led teams can adopt SENAR for AI-native workstreams, planning for coexistence rather than overlay.

Key Differences

Aspect	SAFe	SENAR	Why Different
Production unit	Team (5-9 people)	Supervisor+AI Pair	AI replaces the team as producer
Delivery metric	Velocity (SP/sprint)	Throughput (tasks/session)	AI throughput is unpredictable; SP estimation adds overhead
Time unit	Sprint (2 weeks)	Session (hours)	AI works in bursts, not rhythms
Planning	PI Planning (2 days, full ART)	Increment Planning (1 session)	No multi-team synchronization needed
Quality mechanism	DoR/DoD (team agreements)	Quality Gates (automated code)	AI doesn't feel accountability
Knowledge	CoP, wiki, tribal knowledge	Explicit knowledge base (mandatory)	AI has no long-term memory
Retrospective	Qualitative (feelings + data)	Quantitative (metrics only)	AI work produces precise measurements
Coordination	Scrum of Scrums, ART Sync	Federation Sync	Programmatic dependency tracking
Scaling	Essential → Full → Portfolio SAFe	Core → Team → Enterprise SENAR	Same concept, different production model

Role Mapping

SAFe Role	SENAR Equivalent	Key Difference
Developer	Supervisor	Supervisor directs AI; doesn't write code as primary activity
Product Owner	Context Architect (partial)	CA designs context for AI, not backlog for humans. Business value is shared with Flow Manager
Scrum Master	Flow Manager	FM manages rhythm and cost, not team dynamics. Servant leadership less relevant with AI
QA	Verification Engineer	VE writes acceptance criteria (AI writes tests). Focus shifts to AI-specific defect patterns
RTE	Flow Manager (at Enterprise)	Enterprise FM coordinates multiple Pairs like RTE coordinates teams
System Architect	Chief Supervisor (Enterprise)	Architectural governance across all Pairs

What SAFe Has That SENAR Doesn't

SAFe Element	SENAR Approach
Communities of Practice	Knowledge base replaces human-to-human knowledge sharing with machine-readable entries
Architectural Runway	Knowledge Persistence + Dead End documentation serve similar purpose
IP Iteration	Innovation time recommended in Increment Retrospective
Built-In Quality	Quality Gates (stronger enforcement)
Solution Train / Large Solution	Enterprise configuration with Federation Coordinators (less developed)

Migration from SAFe to SENAR

Organizations moving AI workstreams from SAFe to SENAR should:

1. Start with SENAR Core on one Pair — validate minimum viable process

2. Run SAFe and SENAR in parallel for human-led and AI-led workstreams
3. Track both sets of metrics; compare delivery and quality
4. Expand SENAR to Team when 3+ Pairs are productive
5. SAFe remains for workstreams where humans write the majority of code

---

SENAR Guide: Failure Modes

SENAR can fail. Every methodology can. This document catalogs the ways SENAR implementations break down, how to detect problems early, and what to do about them.

The failure modes are organized into three categories: process failures (the methodology is adopted but practiced incorrectly), organizational failures (the people and structures around SENAR break down), and technical failures (the AI tooling and infrastructure undermines the process).

Process Failures

PF-1: Shadow Coding

Description: Supervisors secretly write code instead of directing AI Agents. The Task appears AI-generated, metrics show normal throughput, but the human is doing the implementation work. This defeats the fundamental premise of SENAR: that the Supervisor's value is judgment, not keystrokes.

Early warning signs:

- Manual Intervention Rate climbs above 30% without corresponding Dead End entries explaining why AI couldn't do the work
- Commit timestamps cluster outside of session windows
- Supervisors report feeling "faster without the AI" but can't articulate what the AI struggled with
- Knowledge base has no Dead End or Gotcha entries — everything "just works" (because the human did it)
- Session tool call counts are suspiciously low relative to task complexity

Recovery action:

1. Conduct a non-punitive audit: compare git diffs against AI session logs for a sample of Tasks
2. Identify which task types trigger shadow coding — these likely have poor context templates

3. Improve context quality for those task types (better AC, patterns, architectural constraints)
4. If AI genuinely cannot handle certain task types, create an explicit "manual" work type with its own tracking

Prevention:

- Normalize manual intervention as a documented, tracked activity — not a shameful secret
- Track Manual Intervention Rate as a diagnostic metric, not a performance metric
- Ensure Supervisors understand that high MIR on certain task types is a signal to improve context, not a personal failure
- Session logs should be reviewed in Quality Sweeps — not to police, but to identify context gaps

---

PF-2: Rubber Stamping

Description: Supervisors accept AI output without meaningful verification. Code passes QG-2 (automated tests, lint, types) but nobody checks whether it actually solves the problem correctly, handles edge cases, or fits the architecture. The human becomes a button-presser.

Early warning signs:

- Defect Escape Rate rises while First-Pass Success Rate stays suspiciously high (everything "passes" but bugs appear later)
- Code review comments are absent or perfunctory ("LGTM", "looks good")
- Verification time per task drops below plausible minimums (a 200-line change reviewed in 30 seconds)
- Architectural drift: patterns diverge across the codebase because nobody enforces consistency
- Security vulnerabilities appear in production that a human reviewer should have caught

Recovery action:

1. Mandatory Quality Sweep focused on the last 2 Increments — compare merged code against requirements
2. Introduce QG-3 (Verification Gate) if not already active: independent review by a different Supervisor
3. Establish minimum review time thresholds proportional to change size

4. Create an "AI Review Checklist" (see Guide 02) and require it to be completed per task

Prevention:

- QG-3 with independent verification is the strongest safeguard
- Rotate Verification Engineer assignments so reviewers stay sharp
- Track Defect Escape Rate prominently — it is the primary indicator of rubber stamping
- Quality Sweeps must sample actual code, not just metrics

PF-3: Gate Bypass Normalization

Description: Quality Gates exist but are routinely bypassed with documented exceptions. "We'll fix it later" becomes the default. Gate bypass was designed for genuine emergencies; when it becomes normal, the gates are decorative.

Early warning signs:

- More than 10% of Tasks have gate bypass entries
- The same gate is bypassed repeatedly (e.g., QG-0 skipped because "we already know what to do")
- Bypass justifications become formulaic ("time pressure", "will address in next increment")
- Tech debt tickets created from bypasses are never completed
- New team members learn to bypass gates as standard practice

Recovery action:

1. Audit all bypass entries from the last 2 Increments — categorize by gate and justification
2. For each repeatedly-bypassed gate, determine: is the gate too strict (adjust criteria) or is the team undertrained (invest in process)?
3. Require bypass approval from someone other than the Supervisor doing the work
4. Create a "bypass budget" per Increment — a maximum number of allowed bypasses. When exhausted, the gate is hard-blocked

Prevention:

- Track bypass rate as a metric reviewed at Increment Retrospectives
- Retrospective must include a "bypass review" — every bypass is discussed, not just counted

- Gate criteria should be reviewed quarterly: if a gate is routinely bypassed, it may need adjustment rather than enforcement
- Distinguish between "gate too strict" (criteria problem) and "discipline too low" (culture problem) — the responses are different

---

PF-4: Knowledge Base Staleness

Description: The Knowledge Base was populated during initial adoption but is no longer maintained. Entries refer to old patterns, deprecated APIs, or architectural decisions that have been superseded. AI Agents receive stale context and produce output based on outdated information.

Early warning signs:

- Knowledge Capture Rate drops below 0.3 entries per task for 3+ consecutive sessions
- New Supervisors report that KB entries contradict what they see in the codebase
- AI Agents produce output using patterns documented in KB that the team has since abandoned
- "Last reviewed" dates on KB entries are older than 3 Increments
- Dead End entries reference tools or libraries that are no longer in the stack

Recovery action:

1. Schedule a dedicated KB audit session — treat it as a Task, not a side activity
2. Mark every entry with a freshness status: current, needs-review, deprecated
3. Delete or archive deprecated entries immediately — stale context is worse than no context
4. Assign KB maintenance as an explicit responsibility (Knowledge Engineer role)

Prevention:

- Quality Sweep ceremony (Standard, Section 7.4) should include KB freshness audit
- Knowledge entries should have a "last verified" timestamp, updated when someone confirms the entry is still accurate
- Set a target Knowledge Capture Rate (0.3-0.5 entries/task) and review at Retrospectives
- When a Task produces output that contradicts a KB entry, the entry must be updated before the Task is marked done

---

PF-5: Dead End Documentation Stops

Description: Teams stop documenting failed approaches. Dead Ends are one of the highest-value knowledge types — they prevent AI Agents from repeating mistakes. When teams are under pressure, Dead End documentation is the first thing dropped because it documents what didn't work, which feels low-priority.

Early warning signs:

- Dead End entries per Increment drop to zero
- AI Agents attempt approaches that were already tried and abandoned in previous sessions
- Session durations increase because the AI is exploring paths that a human remembers failing but didn't document
- Handoff documents mention "we tried X but it didn't work" without creating a KB entry

Recovery action:

1. Review the last 5 session handoffs for mentions of failed approaches — create Dead End entries for each
2. Add "Dead End capture" as an explicit step in the Session End ceremony
3. Make Dead End documentation part of QG-2 acceptance criteria for complex tasks

Prevention:

- Session End template should include a "Failed approaches" section that becomes a Dead End entry
- Knowledge Capture Rate metric should be decomposed by entry type — a healthy KB has a mix of decisions, patterns, gotchas, and dead ends
- Celebrate Dead End documentation: it is not a record of failure, it is a map of the territory

---

PF-6: Session Discipline Erosion

Description: Sessions stretch beyond reasonable limits. Supervisors skip Session Start (no context loading, no task selection) or Session End (no handoff, no metrics capture). The session boundary — SENAR's primary rhythm mechanism — dissolves into continuous, unstructured work.

Early warning signs:

- Session durations exceed 8 hours regularly
- Session Start entries have no task list or context verification
- Handoff documents are empty or missing
- Checkpoints are never taken
- Tool call counts per session exceed 500 (fatigue territory)
- Quality of AI output degrades in the second half of sessions (context window exhaustion)

Recovery action:

1. Enforce hard session time limits — 4-6 hours is the recommended range
2. Make Session End a blocking step: the next session cannot start without a completed handoff
3. Implement automatic checkpoint reminders at 40-60 tool call intervals
4. Review the last 10 sessions for missing handoffs and create them retroactively

Prevention:

- Session Start and Session End are ceremonies, not optional steps
- Tooling should enforce: session cannot start if previous session has no handoff
- Set a maximum tool call budget per session (300-500 depending on task complexity)
- Supervisor fatigue is real: verification quality degrades after 4-6 hours. Process must acknowledge human limits

---

PF-7: Vanity Metrics

Description: Teams optimize metrics without improving outcomes. Throughput is inflated by splitting tasks into trivially small units. First-Pass Success Rate is gamed by lowering acceptance criteria. Lead Time is reduced by starting tasks before they're ready. The numbers look good; the software does not.

Early warning signs:

- Throughput increases dramatically but stakeholder satisfaction doesn't
- Average task complexity drops to "trivial" for 80%+ of tasks
- FPSR is above 95% but Defect Escape Rate is also above 10% (contradictory signals)
- Lead Time decreases but deployed features are incomplete or buggy
- Teams resist adding new metrics or changing existing ones
- Cost per Task drops but total Increment cost doesn't

Recovery action:

1. Cross-reference metrics: Throughput x DER, FPSR x post-deployment defects, Lead Time x rework rate
2. Introduce "outcome metrics" alongside process metrics: deployment frequency, change failure rate, time to restore
3. Redefine task granularity guidelines — a task should represent a meaningful, verifiable change
4. Retrospective should compare metrics against actual stakeholder outcomes

Prevention:

- Never reward or punish based on a single metric
- Metrics exist to diagnose, not to judge — make this explicit in team norms
- DER is the hardest metric to game because it's measured by external reality (production bugs)
- Quality Sweeps should validate that metrics tell a consistent story

---

Organizational Failures

OF-1: Supervisor Burnout

Description: Constant verification of AI output creates cognitive fatigue. The Supervisor must maintain deep technical understanding while processing large volumes of generated code. Unlike traditional development where the developer builds context incrementally, the Supervisor must rapidly context-switch between tasks and verify unfamiliar code patterns.

Early warning signs:

- Verification quality drops in afternoon sessions (rubber stamping after lunch)
- Supervisors report feeling "drained" despite not writing code
- Increased use of gate bypasses in later sessions of the week
- Sick days and turnover increase
- Supervisors start preferring simple tasks and avoiding complex ones

Recovery action:

1. Reduce session length to 3-4 hours with mandatory breaks
2. Rotate between deep verification work and lighter tasks (documentation, knowledge curation)
3. Implement pair supervision: two Supervisors verify each other's complex tasks
4. Ensure Supervisors have autonomy in task selection and session planning

Prevention:

- Design sessions with verification load in mind — not all tasks require the same level of scrutiny
- Budget time for non-verification work: KB maintenance, context improvement, learning
- Track tool calls and session duration — these are leading indicators of burnout
- Respect the difference between "AI can work 24 hours" and "humans cannot verify for 24 hours"

OF-2: Skills Atrophy

Description: Supervisors stop writing code entirely and gradually lose the ability to evaluate AI output at a deep technical level. They can spot surface-level issues (typos, obvious bugs) but miss architectural problems, performance anti-patterns, and subtle security issues. The "judgment over keystrokes" value requires that judgment remains sharp.

Early warning signs:

- Supervisors struggle to explain why a particular implementation approach is wrong
- Architectural decisions are increasingly delegated to the AI with no human challenge
- Code reviews become "does it work?" rather than "is this the right approach?"
- Supervisors avoid tasks in unfamiliar parts of the codebase
- New technologies are adopted without Supervisor understanding of tradeoffs

Recovery action:

1. Institute "manual implementation days" — periodic sessions where Supervisors write code without AI assistance
2. Require Supervisors to write the architectural approach before the AI implements it
3. Rotate Supervisors across different parts of the codebase to force learning
4. Include "explain the implementation" as part of verification: if you can't explain it, you can't verify it

Prevention:

- Supervisors should occasionally implement complex tasks manually to stay sharp
- Technical learning time should be budgeted — not as overhead, but as capability maintenance
- The Context Architect role keeps Supervisors engaged in system design, not just task verification

- Quality Sweeps that require deep code understanding serve double duty: quality assurance and skill maintenance

---

OF-3: AI Tool Vendor Lock-in

Description: The SENAR implementation becomes deeply coupled to a specific AI tool, model, or platform. Session templates, knowledge base format, context injection mechanisms, and verification workflows all assume a particular vendor's API and capabilities. Switching becomes prohibitively expensive.

Early warning signs:

- All context templates use vendor-specific syntax or features
- Knowledge base format is optimized for one model's context window size
- Session management is embedded in a proprietary platform with no export
- Cost tracking depends on vendor-specific billing APIs
- Team expertise is in "using Tool X" rather than "supervising AI agents"

Recovery action:

1. Audit all touchpoints between SENAR process and AI tooling — create an abstraction inventory
2. Define a vendor-neutral interface for context injection, output capture, and session management
3. Extract knowledge base into a portable format (structured markdown, JSON)
4. Run parallel sessions with alternative tools to validate portability

Prevention:

- SENAR deliberately does not prescribe specific AI tools — maintain this neutrality
- Keep knowledge base in a format readable by any LLM (structured text, not proprietary embeddings)
- Session management should be a thin layer, not a platform
- Periodically evaluate alternative tools to maintain optionality

---

OF-4: Over-Process (Bureaucratic SENAR)

Description: SENAR becomes an end in itself. Every task requires exhaustive documentation. Every session has a 30-minute startup ceremony. Every knowledge entry needs three

approvals. The methodology that was designed to increase throughput becomes the primary obstacle to throughput.

Early warning signs:

- Session Start takes longer than 30 minutes
- More time is spent on SENAR ceremonies than on actual work
- Supervisors create "process compliance" tasks that produce no software value
- New Supervisors are overwhelmed by the number of required steps
- Teams maintain two systems: the "official" SENAR process and their actual workflow

Recovery action:

1. Measure "process overhead ratio": time on ceremonies / time on productive work. If it exceeds 20%, the process is too heavy
2. Revisit which SENAR level (Core/Team/Enterprise) actually matches the team's needs — many teams over-adopt
3. Automate everything that can be automated — manual ceremony steps should be minimized
4. Apply SENAR's own principle: "Enforcement over Ceremony" — if a step can be automated, it shouldn't be a manual ceremony

Prevention:

- Start with SENAR Core and upgrade only when specific pain points justify additional process
- Every ceremony should have a defined timebox
- Every manual step should have a justification: "what failure does this prevent?" If the answer is unclear, remove the step
- Periodically run a "process audit" — are all steps still earning their cost?

OF-5: Under-Process (Perpetual L2)

Description: The team adopts SENAR Core (Maturity Level 2) and never progresses. Tasks exist, Start Gate and Done Gate are enforced, FPSR and DER are tracked — but there's no independent verification, no knowledge capture discipline, no federation coordination. The team gets the minimum benefit and plateaus.

Early warning signs:

- Same process for 6+ months with no evolution
- Knowledge base growth has flatlined

- DER is stable but never improves
- Cross-project dependencies are managed ad-hoc (email, chat) rather than through federation
- No Increment Retrospectives — metrics are collected but never reviewed
- Team says "SENAR is working fine" but can't point to specific improvements

Recovery action:

1. Run a maturity assessment across all 6 dimensions (Section 12.2) — identify the weakest
2. Pick ONE dimension to improve in the next Increment — don't try to jump from L2 to L3 across all dimensions simultaneously
3. Assign improvement ownership to a specific person (Flow Manager responsibility)
4. Set a measurable target: "Reduce DER from 12% to 8% by end of Increment"

Prevention:

- Include "process improvement" as a standing item in Increment Planning
- The Flow Manager role exists precisely to drive process evolution — ensure it's staffed
- Set explicit maturity targets with timelines
- Review the SENAR Maturity Model (Section 12) at each Retrospective

---

OF-6: Supervisor Resistance

Description: Developers refuse to adopt the Supervisor role. They see SENAR as de-skilling: taking away the creative, satisfying work of writing code and replacing it with bureaucratic verification of machine output. The resistance may be quiet (passive non-compliance) or loud (active pushback).

Early warning signs:

- High shadow coding rates (PF-1) — developers code and retroactively create tasks to match
- Supervisors describe themselves as "reviewers" or "testers" rather than "engineers"
- Hiring for Supervisor roles is difficult — candidates want "real engineering" positions
- Team morale surveys show dissatisfaction with role definition
- Experienced developers leave for traditional development roles

Recovery action:

1. Address the identity question directly: Supervisor is a more senior role than Developer, not a lesser one

2. Highlight the aspects of the role that require deeper skill: architectural judgment, context design, verification of complex systems
3. Allow Supervisors to choose when to code manually — autonomy reduces resistance
4. Share concrete examples of how the Supervisor role amplifies individual impact (throughput multiplied by AI)

Prevention:

- Frame the transition as career progression, not role diminishment
- Ensure compensation reflects the increased scope and responsibility of the Supervisor role
- Provide the Transition Guide (Guide 04) and allow adequate adjustment time
- Create communities of practice where Supervisors share techniques and celebrate wins

---

OF-7: Knowledge Silos

Description: Each Supervisor develops their own undocumented patterns for context preparation, task structuring, and AI interaction. These patterns are effective but exist only in the Supervisor's head. When they're unavailable, other Supervisors can't replicate their results.

Early warning signs:

- Throughput varies dramatically between Supervisors on similar task types
- Knowledge base entries are sparse or generic — the "real" context is in personal notes
- Supervisors can't cover for each other during absences
- Handoff documents don't capture the "tricks" that make a session productive
- New Supervisors take months to reach baseline productivity

Recovery action:

1. Run "context pairing" sessions: have high-performing Supervisors work alongside others, capturing their undocumented patterns
2. Convert personal notes and templates into shared knowledge entries
3. Standardize context templates for common task types
4. Include "knowledge sharing" as an explicit goal in Quality Sweeps

Prevention:

- Knowledge Capture Rate metric should be reviewed per Supervisor, not just in aggregate
- Context templates should be shared artifacts, not personal files

- Session handoffs should include "what worked well" for context preparation, not just task status
- The Knowledge Engineer role exists to prevent silos — ensure it's actively practiced

---

Technical Failures

TF-1: Model Change Breaks Baselines

Description: The AI model is updated (new version, different provider, or fine-tuning change) and all established baselines become invalid. Throughput, FPSR, Lead Time, and Cost per Task all shift unpredictably. The team loses its ability to detect process problems because the signal is buried in model-change noise.

Early warning signs:

- Sudden shift in all metrics after a model update
- Tasks that previously had high FPSR start failing
- Cost per Task changes significantly (often in both directions for different task types)
- Context templates that worked well now produce worse output
- AI behavior changes subtly: different coding style, different library preferences, different error handling patterns

Recovery action:

1. Treat a model change as a "baseline reset event" — document the change and mark metrics before/after
2. Run a sample of representative tasks on the new model to establish new baselines
3. Update context templates and knowledge entries that assumed specific model behavior
4. Allow 1-2 sessions of reduced throughput expectation while the team calibrates

Prevention:

- Pin model versions in production — don't auto-update
- When evaluating a new model, run a parallel evaluation on a representative task set before switching
- Keep context templates model-neutral: describe what you want, not how the model should produce it
- Maintain a "model change log" that records which model version was used for each Increment's baselines

TF-2: Context Window Limitations

Description: The AI's context window is finite. As projects grow, the context required for a task (codebase knowledge, architectural constraints, related patterns, previous decisions) exceeds what fits in a single session. Tasks are artificially split not because they're naturally separable, but because the AI can't hold enough context.

Early warning signs:

- Tasks are split into sub-tasks that don't make sense as independent units
- AI "forgets" instructions given earlier in the session
- Late-session output contradicts early-session decisions
- Supervisors spend increasing time re-explaining context after checkpoints
- Complex refactoring tasks fail because the AI can't see enough of the codebase simultaneously

Recovery action:

1. Restructure context injection: prioritize high-value context (architectural constraints, active patterns) over low-value context (full file contents)
2. Use knowledge base entries as compressed context: a 5-line decision entry replaces reading 500 lines of code
3. Implement progressive context loading: start with architecture overview, load details as needed
4. Accept that some tasks require multiple sessions — design handoff documents to carry context across the boundary

Prevention:

- Knowledge base entries are context compression tools — invest in high-quality, concise entries
- Architectural documentation should be structured for AI consumption: clear, concise, with explicit constraints
- Design task granularity around natural boundaries, not context window size
- Monitor context window utilization and establish team conventions for context budget allocation

TF-3: Architecturally Wrong but Gate-Passing Code

Description: AI generates code that passes all automated quality gates (tests pass, types check, linter clean, security scan clear) but is architecturally wrong. It might use the wrong pattern, create inappropriate coupling, bypass established abstractions, or introduce a dependency that will cause problems in 6 months. Automated gates verify correctness; they cannot verify judgment.

Early warning signs:

- Code reviews increasingly find "it works but it's wrong" issues
- Similar functionality is implemented differently in different parts of the codebase
- Established abstractions are bypassed — AI creates new patterns instead of using existing ones
- Dependency graph grows in unexpected directions
- Refactoring costs increase over time as architectural drift accumulates

Recovery action:

1. Strengthen the QG-3 Verification Gate with explicit architectural review criteria
2. Create "architectural constraint" knowledge entries that are injected into every session context
3. Run an architectural health audit — identify and document all established patterns
4. Consider architecture-specific linting rules that can be automated (dependency constraints, layer violations)

Prevention:

- Architectural decisions must be documented in the knowledge base, not just in the code
- Context templates for each area of the codebase should include "use this pattern, not that pattern" instructions
- Quality Sweeps should include architectural consistency checks
- The Context Architect role is critical here: they ensure AI receives the right architectural constraints
- Automated architectural fitness functions (dependency rules, layer checks) convert judgment into enforcement

---

TF-4: Phantom Dependencies

Description: Across sessions, AI agents introduce dependencies (libraries, services, API calls, shared state) that are not explicitly tracked. Each dependency is individually reasonable, but

the aggregate creates a fragile system. No single person or document captures the full dependency graph.

Early warning signs:

- Build times increase without obvious cause
- Package manifest (package.json, requirements.txt, go.mod) grows steadily
- "Works on my machine" issues increase — environment setup is unreliable
- Upgrading one dependency triggers cascading failures
- Nobody can explain why a particular dependency was added
- Service coupling increases: changes in one service require changes in others

Recovery action:

1. Run a dependency audit: for every dependency, find the Task that introduced it and verify it's still needed
2. Create a "dependency decision" knowledge entry type — every new dependency requires a documented justification
3. Implement dependency change detection in CI — any new dependency triggers a review flag
4. Prune unused dependencies

Prevention:

- Add dependency justification to QG-2 acceptance criteria: "no new dependencies without documented reason"
- Track dependency count as a secondary metric
- Quality Sweeps should include dependency review
- Context templates should list approved dependencies and their purposes — AI will prefer what it knows about

---

TF-5: Meaningless Test Coverage

Description: AI generates tests that achieve high coverage metrics but don't actually verify meaningful behavior. Tests mirror the implementation: they test that the code does what the code does, not that the code does what the requirements say. This is particularly insidious because the coverage number looks excellent.

Early warning signs:

- Test coverage is above 90% but Defect Escape Rate is also high

- Tests break when implementation changes but not when behavior changes (brittle, tightly coupled tests)
- Test descriptions are generic: "should work correctly", "handles input"
- Tests don't cover edge cases, error paths, or boundary conditions
- Mutation testing kills few mutants despite high line coverage
- Integration tests are actually unit tests with mocked-out dependencies

Recovery action:

1. Run mutation testing to evaluate actual test effectiveness — coverage means nothing without kill rate
2. Review test quality in Quality Sweeps: do tests verify requirements or mirror implementation?
3. Require acceptance criteria to include specific test scenarios — AI writes tests from scenarios, not from code
4. Introduce property-based testing for critical paths

Prevention:

- Acceptance criteria should define test cases: "given X, when Y, then Z" — not "write tests for this function"
- Track mutation testing score alongside coverage
- QG-2 should include test quality checks, not just coverage thresholds
- Knowledge base should contain testing patterns: "how we test X-type functionality"
- Separate the test-writing agent prompt from the implementation agent prompt — tests should verify requirements, not mirror code

---

Summary

No failure mode in this document is theoretical. Every one has been observed in practice, either in AI-native development or in analogous patterns from traditional Agile/SAFe adoption.

The common thread across all failure modes is the same: **when the uncomfortable parts of the process are skipped, the process stops working.** Verification is uncomfortable. Documentation is tedious. Dead Ends feel like wasted time. Gate compliance feels like bureaucracy. But these are the load-bearing elements of SENAR — remove them and the structure collapses.

The best defense is not more process but better feedback loops:

1. **Metrics that cross-reference** — no single metric tells the truth alone

2. **Quality Sweeps that sample reality** — not just metrics dashboards but actual code and actual knowledge entries
3. **Retrospectives that ask hard questions** — not "are our numbers good?" but "is our software good?"
4. **A culture where documenting failure is valued** — Dead Ends and failure mode awareness prevent repeated mistakes

---

SENAR Guide: Requirements Engineering for AI-Native Development

Why Requirements Matter More with AI

In traditional development, a vague requirement leads to a conversation: "Did you mean X or Y?" The developer asks, clarifies, and implements correctly.

With AI, a vague requirement leads to a confident, plausible implementation of the wrong thing. AI does not ask clarifying questions — it picks an interpretation silently. The resulting code compiles, passes the tests the AI wrote (which test the wrong behavior), and looks correct on review.

The cascade principle: A defect at the requirement level is the most expensive defect. It produces correct-looking code that solves the wrong problem, correct-looking tests that verify the wrong behavior, and correct-looking documentation that describes the wrong system. Every downstream artifact inherits the original defect.

This is why SENAR's first value is "Context over Code" — and requirements are the most important context.

The Three Levels

SENAR defines three requirement levels. Not all are needed for every task — depth scales with complexity and team size.

BR — Business Requirement

What: A stakeholder-level need expressed in business terms. **Who writes it:** Stakeholder, Product Owner, or Context Architect. **Where it lives:** Increment objective, Epic goal, or dedicated BR entry in the knowledge base.

Examples:

- "Users must be able to reset their password without contacting support"
- "The system must process orders within 30 seconds during peak load"
- "API must support OAuth 2.0 for third-party integrations"

Properties: A good BR is:

- Business-oriented (no implementation details)
- Verifiable (you can determine if the system satisfies it)
- Independent (achievable without solving another BR first, or dependency is explicit)

SR — System Requirement

What: A system-level capability derived from a BR. Describes what the system must do, not how. **Who writes it:** Context Architect (Team+) or Supervisor (Core). **Where it lives:** Story goal, or dedicated SR entry linked to parent BR.

Examples (derived from BR "password reset"):

- "POST /auth/reset-password sends a reset link to the user's email"
- "Reset link expires after 1 hour"
- "User can set a new password using a valid reset token"
- "System logs all password reset attempts for security audit"

Properties: A good SR is:

- Specific enough to decompose into Tasks
- Traceable to a parent BR
- Testable at the system level (integration test or E2E test)

TR — Task Requirement

What: An implementation-level requirement. Corresponds directly to a Task's goal and acceptance criteria. **Who writes it:** Supervisor. **Where it lives:** Task goal + acceptance criteria fields.

Example (derived from SR "POST /auth/reset-password"):

- **Goal:** Implement POST /auth/reset-password endpoint
- **AC:**
    1. Returns 200 and sends email for existing user
    2. Returns 200 for non-existing user (no information leak)
    3. Returns 422 if email field missing
    4. Reset token is cryptographically random, 32 bytes
    5. Token stored hashed in database (not plaintext)

6. Rate limited to 3 requests per email per hour

Properties: A good TR (= good acceptance criteria) is:

- Independently verifiable (each AC can be tested separately)
- Includes at least one negative scenario (error case)
- Specifies behavior, not implementation ("returns 422" not "use Pydantic validator")

Test Model (TM) — the Verification Bridge

The Test Model is NOT a fourth requirement level. It is a verification artifact derived from Task Requirements.

```
BR → SR → TR → [TM] → Code + Tests
                ↑ requirement    ↑ verification artifact
```

What a Test Model contains:

- Test cases for each AC (what to test)
- Test data (boundary values, error inputs)
- Expected results (what "pass" looks like)
- Verification method (automated unit test, integration test, manual demo, measurement)

In AI-native development, AI typically generates tests alongside code. The danger: AI generates tests that pass but don't actually exercise the stated AC. The Test Model is the Supervisor's check that generated tests match the requirements, not just achieve coverage.

Configuration	Test Model Discipline
Core	Implicit — Supervisor reviews generated tests against AC
Foundation	Implicit — same as Core; combined Supervisor role performs AC review
Team	SHOULD be explicitly reviewed — each AC has a corresponding test
Enterprise/Regulated	SHALL be documented, traceable to TRs, independently verifiable

Why not a requirement level? Test cases describe *how to verify*, not *what the system must do*. This follows ISO/IEC 29119 (software testing) which separates test design from requirements. Making TM a requirement level would force Core users to manage 4 levels — overhead that kills adoption.

Requirement Quality Properties

Property	Definition	Core	Team+
Verifiability	Each requirement can be tested, measured, or demonstrated	SHALL	SHALL
Consistency	No contradictions between requirements at same or higher level	SHOULD	SHALL
Sufficiency	The set of child requirements fully covers the parent	SHOULD	SHALL
Non-redundancy	No duplicate requirements across Stories	SHOULD	SHALL
Traceability	Every TR traces up to a BR; every BR decomposes down to TRs	SHOULD	SHALL
Reusability	Verified requirements can be parameterized for similar tasks	MAY	SHOULD

Verifiability in Practice

Bad: "The system should be fast" — fast by whose standard? Good: "API response time < 200ms at p95 under 100 concurrent users"

Bad: "Error handling should be robust" — what does robust mean? Good: "All API endpoints return structured error responses with HTTP status code, error code, and human-readable message"

Bad: "The UI should be user-friendly" — unmeasurable. Good: "A new user can complete the registration flow in under 2 minutes without external help"

AC Templates for Common Tasks

REST API Endpoint

1. Returns expected status code and body for valid input
2. Returns 401/403 for unauthorized/forbidden access

3. Returns 422 with descriptive error for invalid input
4. Returns 404 for non-existent resource
5. Response matches documented schema
6. Rate limiting applied per configuration

Database Migration

1. Migration is idempotent (safe to run multiple times)
2. Rollback migration exists and works
3. No data loss during migration
4. Migration completes within acceptable time for production data volume
5. Indexes created for new query patterns

UI Component

1. Renders correctly in target viewport range (320–1920px)
2. Accessible: keyboard navigable, screen reader compatible (WCAG AA)
3. Handles loading, empty, and error states
4. Responsive to theme/dark mode if applicable

Integration / External API

1. Handles success response correctly
2. Handles timeout (configurable, default 5s)
3. Handles error response with structured error
4. Retry policy for transient failures
5. Credentials not hardcoded (env/config)

Infrastructure / DevOps

1. Configuration change is reversible
2. Health check endpoint updated if applicable
3. Monitoring/alerting covers the change
4. Documentation updated (runbook, architecture diagram)

Anti-Patterns

Specification Vacuum

Problem: Task has a goal but no acceptance criteria. AI produces something that "looks right" but nobody can verify it. **Fix:** Every Task SHALL have acceptance criteria (QG-0). No AC = no start.

Over-Specification

Problem: 20 acceptance criteria for a trivial task. Overhead exceeds value. **Fix:** Match AC depth to task complexity. Trivial: 2–3 AC. Complex: 5–8 AC. If you need 15+ AC, the task should be split.

Copy-Paste AC

Problem: Same AC copied across tasks without adaptation. "Returns 200" everywhere, but some endpoints should return 201 or 204. **Fix:** Use templates as starting points, then customize. Templates save time; blind copying introduces bugs.

Untraceable Requirements

Problem: Tasks exist without any link to why they exist. When priorities change, nobody knows which tasks to drop. **Fix:** Every Task links to a Story or BR (Rule 9.10). If you can't explain why a task exists, it probably shouldn't.

Ambiguous Criteria

Problem: "System should handle edge cases properly." Which edge cases? What is "properly"? **Fix:** Name the specific edge cases. "System returns 409 when duplicate email. System returns 413 when file exceeds 10MB. System returns 429 when rate limit exceeded."

Scaling by Configuration

Aspect	Core (1–2 Pairs)	Foundation (1–3 Pairs)	Team (3–10 Pairs)	Enterprise (10+)
Hierarchy depth	BR → TR (2 levels)	BR → TR (2 levels)	BR → SR → TR (3 levels)	Full + test derivation
Who manages	Supervisor	Supervisor (combined roles)	Context Architect	Context Architect + review
Storage	Task fields	Task fields + KB entries	Knowledge Base	Versioned + auditable
Reuse	Informal patterns	Informal patterns	KB templates	Cross-project library
Traceability	SHOULD	SHOULD	SHALL	SHALL + audit trail
Quality review	Self-review	Monthly Quality Sweep	QG-1	QG-1 + independent review
Overhead per task	~1 min (write AC)	~1–2 min (write AC + log)	~3 min (decompose + link)	~5 min (full traceability)
Test Model	Implicit (review tests vs AC)	Implicit (review tests vs AC)	Explicit review	Formal TM documentation

Requirements-as-Code (Enterprise)

At Enterprise scale, requirements are managed with the same discipline as code: versioned, reviewed, tested by CI.

What This Means

```
requirements/
  BR-001-oauth.md          # Business Requirement
  SR-001-google-oauth.md   # System Requirement (links to BR-001)
  SR-002-token-refresh.md  # System Requirement (links to BR-001)
```

Each requirement file contains:

- Unique ID and title
- Level (BR/SR)
- Parent link (SR → BR)
- Status (draft/approved/verified/deprecated)
- Version history (git log)
- Child links (BR → list of SRs, SR → list of Task slugs)

Task Requirements (TR) remain in the task tracker as goal + AC — they don't need separate files because they ARE the task.

CI Checks for Requirements

Check	What It Catches
Orphan detection	BRs with no child SRs; SRs with no Tasks
Traceability validation	Tasks without requirement links; broken parent references
Status consistency	Tasks referencing `deprecated` requirements
Coverage report	% of BRs fully decomposed and implemented
Change impact	When a BR changes, list all affected SRs and Tasks

Requirement Library and Reuse

Verified requirements (status: `verified`, used successfully in N projects) become library entries:

```
library/
  patterns/
    rest-api-endpoint.md    # Template: AC for any REST endpoint
    db-migration.md         # Template: AC for any migration
    auth-integration.md     # Template: AC for auth flows
```

A new project imports a library pattern, parameterizes it, and gets proven AC that have been tested across multiple implementations. This directly improves First-Pass Success Rate — AI receives battle-tested context instead of first-draft requirements.

When to Adopt

Requirements-as-code adds overhead. It pays off when:

- 10+ Supervisor+AI Pairs share requirements across projects
- Regulatory compliance requires audit trails (ISO 9001, ASPICE, medical devices)
- Cross-project requirement reuse is a strategic goal
- Requirement changes are frequent and impact analysis is needed

For SENAR Core and most Team configurations, requirements in the task tracker and Knowledge Base are sufficient.

SENAR Guide: Adopting SENAR in Legacy Codebases

Most software is not greenfield. You have 200,000 lines of undocumented code, and Rule 9.11 says "documentation sufficient for AI to understand module purpose." You are not going to document everything before starting. Here's how to adopt SENAR incrementally.

Principle: Next Task, Not Full Retrofit

You don't need to "SENAR everything." Start with the next task. When that task touches a module, document that module. When the next task touches another module, document that one. Over time, the documentation frontier advances with actual work.

What NOT to do: Don't create a "documentation sprint" that takes 3 weeks and produces docs nobody reads. Documentation written in isolation from implementation is outdated before it's finished.

Phase 1: Start with Habits (Day 1)

Adopt the 6 Quick Start habits immediately. They require zero codebase preparation:

1. Write goal + AC before each AI task (even in legacy code)
2. Set scope boundaries ("change ONLY this file, don't refactor the module")
3. Verify against AC, not intuition
4. Document dead ends (especially important in legacy — "tried X, failed because Y")
5. Run tests
6. Capture knowledge

Scope boundaries (habit 2) are critical for legacy: AI agents will try to "improve" surrounding code if you don't fence them.

Phase 2: Document on Contact (Week 1+)

Every time a task touches a module, add minimum documentation:

```
"""
Module: user_auth
Purpose: Handles login, registration, and session management.
Public API:
  - login(email, password) -> Session
  - register(email, password, name) -> User
  - verify_session(token) -> User | None
Dependencies: database (PostgreSQL), redis (session store)
Boundaries: Does NOT handle OAuth (see oauth_provider module).
"""
```

This takes 5 minutes per module. It satisfies Rule 9.11 at the SENAR Core level. From this point forward, AI tasks touching this module need less explicit context in the Task goal — the docstring provides it.

Key: Write for AI, not for humans. The AI doesn't care about your design philosophy. It needs: what this module does, what the public interface is, what it connects to, and what it does NOT do.

Phase 3: Build Knowledge Base from Tribal Knowledge (Week 2+)

Legacy codebases have tribal knowledge — things people know but haven't written down. As you encounter these during SENAR tasks:

- **Dead End:** "Can't use async in the auth module because it depends on a sync middleware chain" → document it
- **Gotcha:** "The order status field uses integers 1-7, not the enum — legacy migration never happened" → document it
- **Decision:** "We use raw SQL instead of ORM for the reporting module because of the complex join queries" → document it

These become Knowledge Base entries that AI reads in future sessions. Every documented gotcha prevents one future $105 escaped defect.

Phase 4: Requirement Links for New Work (Month 1+)

New features on legacy codebases often lack clear requirements — someone says "fix the thing" and you investigate. SENAR's Exploration (Section 6.1) handles this:

1. Start an Exploration (time-bounded investigation)
2. When you understand what needs to happen, create a Task with goal + AC
3. Link the Task to a Story or BR — even if the BR is just "reduce support tickets about X"

For bug fixes in legacy code:

- BR: the original business need that the bug violates (e.g., "users must be able to reset passwords")
- TR: the specific fix with AC (e.g., "POST /reset-password returns 200 for non-existing emails — no information leak")

Phase 5: Quality Gates on Legacy (Month 2+)

QG-0 works immediately on legacy: every task has goal + AC before starting.

QG-2 may need adaptation:

- "CI passes" requires that CI exists. If not, add minimal CI as a one-time investment.
- "Tests pass" requires tests. For untested legacy modules, the minimum is: add tests for the code you change. Don't test what you don't touch.
- "Types clean" may not apply to dynamically typed legacy. Use what's available (mypy for Python, TypeScript strict for JS migrations).

QG-3 and QG-4 are Team+ and Enterprise — defer until SENAR Core is solid.

What You Get

After 3 months of SENAR Core on a legacy codebase:

- Documentation frontier advancing with actual work (not rotting in a wiki)
- Knowledge base of gotchas that prevent repeated failures
- Measurable FPSR showing whether context quality is improving

- Dead ends that save hours per session
- No "big bang" documentation effort, no process disruption

Anti-Pattern: The Documentation Sprint

Problem: Manager says "let's spend 2 weeks documenting everything before we start using AI." **Why it fails:** Documentation without implementation context is abstract and immediately stale. The person documenting module X hasn't worked in it for 6 months — they'll miss the gotchas. **SENAR approach:** Document on contact. The person documenting module X is the person implementing a task in module X right now. They know exactly what AI needs to know because they just discovered it.

SENAR Guide: A Complete Task, Start to Finish

This is not a diagram. This is one task, from requirement to done, exactly as it happens in practice. The task: add a password reset endpoint to a REST API.

Stack: Python 3.14, FastAPI, PostgreSQL, pytest. Configuration: SENAR Core (solo Supervisor).

---

1. The Requirement

The product backlog has a business requirement:

> ***BR-042:*** *Users must be able to reset their passwords without contacting support.*

The Supervisor decomposes this into a Task Record:

```
task: impl-password-reset
story: user-authentication
goal: >
  Implement POST /auth/password-reset/request and
  POST /auth/password-reset/confirm endpoints that allow
  users to reset their password via a time-limited token.
acceptance_criteria:
  1. POST /auth/password-reset/request with valid email returns 202
and creates a reset token
  2. POST /auth/password-reset/request with unknown email returns 202
(no information leak)
  3. Reset token expires after 30 minutes
  4. POST /auth/password-reset/confirm with valid token + new
password updates the password and invalidates the token
  5. POST /auth/password-reset/confirm with expired or invalid token
returns 400
  6. POST /auth/password-reset/confirm with password shorter than 8
chars returns 422
role: supervisor
complexity: M
```

Time spent: ~2 minutes.

---

2. Start Gate (QG-0 in the Standard) Check

Before the task goes to `active`, the system checks Context Gate criteria:

Criterion	Status
Goal is non-empty	**PASS** — goal describes both endpoints and their purpose
Acceptance criteria exist	**PASS** — 6 criteria, each independently testable
Story link set	**PASS** — linked to `user-authentication`
Role assigned	**PASS** — `supervisor`

Start Gate (QG-0): **PASS**. Task moves to `active`.

---

3. The AI Prompt

The Supervisor opens the terminal and gives this prompt to the AI agent:

```
Implement password reset for our auth module.

Goal: POST /auth/password-reset/request and POST /auth/password-
reset/confirm
endpoints. Request creates a time-limited token. Confirm validates the
token
and updates the password.

Acceptance criteria:
1. POST /request with valid email → 202, creates reset token
2. POST /request with unknown email → 202 (no information leak)
3. Token expires after 30 minutes
4. POST /confirm with valid token + new password → updates password,
invalidates token
5. POST /confirm with expired/invalid token → 400
6. POST /confirm with password < 8 chars → 422

Scope:
- Change ONLY files in src/auth/ and tests/auth/
- Follow the patterns in src/auth/router.py and src/auth/service.py
- Do NOT modify the User model — add a new PasswordResetToken model
- Use the existing get_password_hash() for hashing
- Do NOT add new dependencies without asking

Plan:
1. Create PasswordResetToken model in src/auth/models.py
2. Add reset request/confirm logic in src/auth/service.py
3. Add endpoints in src/auth/router.py
4. Add schemas in src/auth/schemas.py
5. Write tests for all 6 AC
```

Time spent: ~1 minute (most of it was already in the task record).

4. The AI Output

The AI generates four files:

`src/auth/models.py` — adds `PasswordResetToken` model:

```
class PasswordResetToken(Base):
    __tablename__ = "password_reset_tokens"

    id: Mapped[int] = mapped_column(primary_key=True)
    user_id: Mapped[int] = mapped_column(ForeignKey("users.id"))
    token: Mapped[str] = mapped_column(String(64), unique=True,
index=True)
    expires_at: Mapped[datetime] = mapped_column()
    used: Mapped[bool] = mapped_column(default=False)
```

`src/auth/schemas.py` — adds request/response schemas:

```
class PasswordResetRequest(BaseModel):
    email: EmailStr

class PasswordResetConfirm(BaseModel):
    token: str
    new_password: str = Field(min_length=8)
```

`src/auth/service.py` — adds two service functions:

```python
from datetime import timezone

async def request_password_reset(db: AsyncSession, email: str) ->
None:
    user = await get_user_by_email(db, email)
    if user is None:
        return  # AC-2: no information leak
    token = secrets.token_urlsafe(48)
    expires_at = datetime.now(timezone.utc) + timedelta(minutes=30)
# AC-3
    reset = PasswordResetToken(user_id=user.id, token=token,
expires_at=expires_at)
    db.add(reset)
    await db.commit()


async def confirm_password_reset(
    db: AsyncSession, token: str, new_password: str
) -> None:
    reset = await db.execute(
        select(PasswordResetToken).where(
            PasswordResetToken.token == token,
            PasswordResetToken.used == False,
        )
    )
    reset = reset.scalar_one_or_none()
    if reset is None or reset.expires_at <
datetime.now(timezone.utc):
        raise HTTPException(status_code=400, detail="Invalid or
expired token")
    user = await db.get(User, reset.user_id)
    user.password_hash = get_password_hash(new_password)
    reset.used = True  # AC-4: invalidate token
    await db.commit()
```

`src/auth/router.py` — adds two endpoints:

```python
@router.post("/password-reset/request", status_code=202)
async def password_reset_request(
    body: PasswordResetRequest, db: AsyncSession = Depends(get_db)
):
    await request_password_reset(db, body.email)
    return {"detail": "If the email exists, a reset link has been sent"}


@router.post("/password-reset/confirm", status_code=200)
async def password_reset_confirm(
    body: PasswordResetConfirm, db: AsyncSession = Depends(get_db)
):
    await confirm_password_reset(db, body.token, body.new_password)
    return {"detail": "Password updated"}
```

`tests/auth/test_password_reset.py` — six tests:

```python
async def test_request_reset_valid_email(client, db, user):
    """AC-1: valid email returns 202, creates token."""
    resp = await client.post("/auth/password-reset/request", json=
{"email": user.email})
    assert resp.status_code == 202
    token = await db.execute(select(PasswordResetToken).where(
        PasswordResetToken.user_id == user.id
    ))
    assert token.scalar_one_or_none() is not None


async def test_request_reset_unknown_email(client):
    """AC-2: unknown email returns 202 (no leak)."""
    resp = await client.post("/auth/password-reset/request",
                             json={"email": "nobody@example.com"})
    assert resp.status_code == 202


async def test_token_expires_after_30_minutes(client, db, user):
    """AC-3: token created with 30-minute expiry."""
    await client.post("/auth/password-reset/request", json={"email":
user.email})
    token = (await
db.execute(select(PasswordResetToken))).scalar_one()
    delta = token.expires_at - datetime.now(timezone.utc)
    assert 29 * 60 < delta.total_seconds() < 31 * 60


async def test_confirm_reset_valid_token(client, db, user,
reset_token):
    """AC-4: valid token + new password → password updated, token
invalidated."""
    resp = await client.post("/auth/password-reset/confirm",
                             json={"token": reset_token.token,
"new_password": "newsecure123"})
    assert resp.status_code == 200
    await db.refresh(reset_token)
    assert reset_token.used is True
    await db.refresh(user)
    assert verify_password("newsecure123", user.password_hash)
```

```
async def test_confirm_reset_expired_token(client,
expired_reset_token):
    """AC-5: expired token returns 400."""
    resp = await client.post("/auth/password-reset/confirm",
                             json={"token":
expired_reset_token.token, "new_password": "newsecure123"})
    assert resp.status_code == 400


async def test_confirm_reset_short_password(client, reset_token):
    """AC-6: password < 8 chars returns 422."""
    resp = await client.post("/auth/password-reset/confirm",
                             json={"token": reset_token.token,
"new_password": "short"})
    assert resp.status_code == 422
```

---

5. Verification Against AC

The Supervisor does not skim the code and say "looks good." The Supervisor goes through the acceptance criteria one by one:

AC	What to Check	Verdict
1. Valid email → 202 + token created	`test_request_reset_valid_email` asserts 202 and token exists in DB	**PASS**
2. Unknown email → 202	`test_request_reset_unknown_email` asserts 202, no error	**PASS**
3. Token expires in 30 min	`test_token_expires_after_30_minutes` checks `expires_at` delta	**PASS**
4. Valid confirm → updates password, invalidates token	`test_confirm_reset_valid_token` checks `used=True` and password changed	**PASS**
5. Expired/invalid token → 400	`test_confirm_reset_expired_token` asserts 400	**PASS**
6. Short password → 422	`test_confirm_reset_short_password` asserts 422	**PASS**

Every criterion has a corresponding test. Every test asserts the expected behavior, not the implementation.

Time spent: ~3 minutes reading tests and confirming the mapping.

---

6. A Dead End

But first — something went wrong during execution.

The AI's first attempt used `bcrypt` for the password hashing utility it added alongside the endpoint:

```
from bcrypt import hashpw, gensalt

def get_password_hash(password: str) -> str:
    return hashpw(password.encode(), gensalt()).decode()
```

When tests ran:

```
E   ModuleNotFoundError: No module named '_bcrypt'
```

The `bcrypt` package has a C extension that does not compile on Python 3.14 (as of March 2026). The Supervisor directed the AI to switch:

```
bcrypt import fails on Python 3.14. Switch to argon2-cffi with argon2id
hasher.
The existing get_password_hash() already uses argon2 — don't create a new
one,
use the existing function.
```

The AI corrected to use the existing `get_password_hash()` from `src/auth/utils.py` (which already uses `argon2-cffi`). Tests pass.

The Supervisor writes a dead end entry:

```
Dead end: bcrypt package — C extension fails to compile on Python 3.14.
Switched to argon2-cffi (argon2id). The existing get_password_hash()
utility
already uses argon2id. Don't add bcrypt to new endpoints.
```

Time spent: ~30 seconds to write the entry.

---

7. Done Gate (QG-2 in the Standard) Check

The Supervisor runs the Implementation Gate:

```
$ pytest tests/auth/test_password_reset.py -v
======================== 6 passed in 2.14s ========================

$ mypy src/auth/ --strict
Success: no issues found in 6 source files

$ ruff check src/auth/
All checks passed!
```

Criterion	Status
CI pipeline passes	**PASS** — all green
All tests pass	**PASS** — 6/6
Type checking clean	**PASS** — mypy strict, zero errors
No new linting violations	**PASS** — ruff clean
Acceptance criteria verified	**PASS** — all 6 AC checked in Section 5
No security vulnerabilities	**PASS** — `argon2id` is OWASP-recommended; no new deps with CVEs

Done Gate (QG-2): **PASS**. Task moves to `done`.

Time spent: ~2 minutes (mostly waiting for CI).

> ***Measuring FPSR in practice:*** *Count a task as 'first-pass success' if it passes QG-2 without returning to active status. Track via your task management tool: tasks that go planning→active→done = success; tasks that go active→done→active→done = rework. FPSR = successes / total completed × 100%.*

---

8. Knowledge Capture

The Supervisor creates a knowledge entry for the decision made during this task:

```
type: decision
title: "Using argon2id for all password hashing"
context: >
  Password reset endpoint needed hashing. bcrypt fails on Python
3.14.
  argon2id is the OWASP recommendation and already used in
get_password_hash().
decision: >
  All password operations use argon2-cffi with argon2id algorithm via
  the shared get_password_hash() utility in src/auth/utils.py.
  Do not introduce bcrypt or any other hashing library.
related_dead_end: "bcrypt C extension incompatible with Python 3.14"
```

Time spent: ~30 seconds.

---

9. Session End Snippet

At session end, the handoff captures everything a future session (or a different Supervisor) needs:

```
Session #12 — 2026-03-22 — 95 minutes — 4 tasks done

Completed:
  - impl-password-reset (this task)
  - impl-email-verification
  - fix-jwt-refresh-race
  - impl-logout-endpoint

Dead ends:
  - bcrypt C extension fails on Python 3.14 → use argon2-cffi

Knowledge entries:
  - decision: argon2id for all password hashing (OWASP, Python 3.14
compat)
  - gotcha: JWT refresh token race condition — need DB-level locking

Next session:
  - impl-oauth-google — Google OAuth integration
  - impl-rate-limiting — rate limit on auth endpoints (5 req/min)

Warnings:
  - OAuth requires new env vars (GOOGLE_CLIENT_ID, GOOGLE_CLIENT_SECRET)
    — add before starting the task
```

What This Cost

Step	Time
Writing goal + acceptance criteria	~2 min
Writing the AI prompt	~1 min
AI generates code + tests	~3 min (waiting)
Verification against AC	~3 min
Dead end documentation	~30 sec
Knowledge entry	~30 sec
Total	**~10 min**

Without AI, this task — two endpoints, a model, schemas, six tests, hashing research — would take approximately 25–40 minutes of manual coding.

The overhead of SENAR discipline (goal, AC, verification, knowledge capture) adds roughly 4 minutes to a task. The AI generates the remaining 6 minutes of work. The net result: a fully tested, documented, traceable endpoint in under 10 minutes.

But the real savings are not in this session. They are in the next session, when someone (or an AI agent) needs to add another auth endpoint and finds:

- The dead end that prevents them from wasting 15 minutes on bcrypt.
- The decision that tells them to use `get_password_hash()`.
- The acceptance criteria pattern they can copy for the next endpoint.

That future time saved is where SENAR pays for itself many times over.

Key Takeaways

1. **The AC did the heavy lifting.** Six criteria, written in 2 minutes, drove the entire implementation — what the AI generated, what the tests checked, what the Supervisor verified. Without them, the Supervisor would have been reading code and guessing whether it's correct.

2. **The dead end is an investment, not overhead.** Thirty seconds of typing saves the next person (or AI) from repeating the same failure. In a knowledge base with 100+ dead ends, this compounds into hours saved per increment.

3. **Done Gate (QG-2) is not a ceremony.** It's your test runner, type checker, linter, and a verification table — the specific tools depend on your stack. It takes 2 minutes. If any line is red, the task is not done. No judgment calls, no "it's probably fine."
4. **The handoff makes sessions independent.** Session #13 can be started by a different person, a different AI agent, or the same Supervisor after a weekend. Everything needed is in the handoff — no tribal knowledge, no "let me remember where I was."

A Note on This Example

This walkthrough shows a clean-pass scenario: one dead end, one retry, everything green on the second attempt. Real sessions are often messier — multiple dead ends, partial test failures, AC that turn out ambiguous mid-implementation, unexpected dependency conflicts. This example demonstrates the SENAR workflow structure, not the typical difficulty level. Your sessions will vary.

Stack Variations

The practices above apply identically regardless of technology stack. Here are the same acceptance criteria patterns for common stacks:

> ***Note:*** *Stack variations illustrate framework-specific patterns, not exact translations of the Python example. Adapt acceptance criteria to your stack's conventions.*

Java / Spring Boot

Task: Implement password reset endpoint **AC:**

1. POST /api/v1/auth/reset-password accepts {email} → returns 200 (no email enumeration)
2. Token stored with BCrypt hash, expires in 1 hour
3. Rate limited: 3 requests per email per hour (@RateLimiter annotation or filter)
4. Integration test with @SpringBootTest and MockMvc
5. Negative: invalid token returns 400, expired token returns 410

Go / Gin

Task: Implement password reset endpoint **AC:**

1. POST /api/v1/auth/reset-password accepts {email} → returns 200
2. Token stored with bcrypt.GenerateFromPassword, 1h expiry
3. Rate limited: middleware with sync.Map or Redis
4. Table-driven tests with testify
5. Negative: invalid/expired token returns appropriate status

TypeScript / NestJS

Task: Implement password reset endpoint **AC:**

1. POST /api/v1/auth/reset-password accepts {email} → returns 200
2. Token stored with bcrypt.hash, 1h TTL
3. @Throttle() decorator or custom guard
4. Jest e2e test with supertest
5. Negative: ValidationPipe rejects malformed input

Full worked examples for Java/Spring Boot, Go, and TypeScript/NestJS are planned for the SENAR Guide v1.4.

SENAR Guide: Tool Integration

SENAR is tool-agnostic — the standard never requires a specific product. But the practices map differently to each tool. This chapter shows how to apply SENAR with the three most common AI coding tools, plus a minimal knowledge base setup for teams with no infrastructure.

SENAR with Claude Code

Claude Code is a terminal-based AI agent with persistent memory, slash commands, and MCP (Model Context Protocol) integration. It is the closest match to SENAR's assumptions about AI agents.

Goal + AC → Task Description in Prompt

Give the goal and acceptance criteria directly in the prompt. Claude Code processes structured text well:

```
Implement POST /orders/{id}/cancel endpoint.

Goal: Allow users to cancel their own orders if the order is in "pending"
status.

Acceptance criteria:
1. POST /orders/{id}/cancel with valid order in "pending" → 200, status
changes to "cancelled"
2. POST /orders/{id}/cancel with order in "shipped" → 409 Conflict
3. POST /orders/{id}/cancel on another user's order → 403
4. Cancelled order cannot be cancelled again → 409
5. Returns 404 for non-existent order ID

Scope: change ONLY src/orders/ and tests/orders/. Follow patterns in
src/orders/router.py.
```

Scope Boundaries → CLAUDE.md + Slash Commands

`CLAUDE.md` at the project root defines persistent boundaries the agent reads on every session:

```
# Boundaries
- NEVER modify files outside src/ and tests/
- NEVER change database migration files directly — use alembic
- Follow existing patterns in router.py, service.py, schemas.py
- Do NOT add dependencies without asking
```

Use `/plan` before complex tasks to have the agent propose a plan for review before executing.

Knowledge Base → CLAUDE.md + Memory System

Claude Code has two knowledge layers:

Layer	SENAR Mapping	Persistence
`CLAUDE.md`	Project-level knowledge (rules, patterns, boundaries)	In repo, version-controlled
Memory files (`~/.claude/projects/*/memory/`)	Dead ends, decisions, gotchas	Per-user, persists across sessions

When you hit a dead end, tell Claude Code explicitly:

```
Remember this: bcrypt fails on Python 3.14. Always use argon2-cffi.
```

It writes to its memory system. Future sessions read this automatically.

For teams using MCP, knowledge entries can be pushed to a shared knowledge base:

```
Create a knowledge entry: "argon2id for password hashing — bcrypt incompatible
with Python 3.14. OWASP recommendation. Use get_password_hash() from
auth/utils.py."
```

Dead Ends → Knowledge Entries via Memory or MCP

Immediate capture during the session:

```
That approach failed. Remember: SQLAlchemy async sessions cannot use lazy
loading. Must use selectinload() or joinedload() for relationships.
```

QG-0 → /plan Skill

Before starting a task, use `/plan` to validate that the task has sufficient context:

```
/plan — Review this task before implementation. Check that goal, AC, and
scope
are clear. Flag anything ambiguous.
```

The agent reviews the task description and flags missing AC, ambiguous scope, or unstated assumptions.

QG-2 → /review Skill

After implementation, use `/review` to run the Implementation Gate:

```
/review — Check this implementation against the acceptance criteria.
Verify: tests exist for each AC, mypy clean, no linting violations.
```

Session Discipline

Claude Code tracks tool call counts internally. Use checkpoints:

```
/checkpoint — Save progress. 47 tool calls since last checkpoint.
```

Session handoffs go into the memory system or a handoff file the agent reads on next startup.

SENAR with Cursor

Cursor is a VS Code fork with inline AI chat, file-level context via `@mentions`, and project-level rules via `.cursorrules`.

Goal + AC → .cursorrules + Prompt Structure

Create a `.cursorrules` file with persistent instructions:

```
# Project Rules
- Every implementation MUST have tests for all acceptance criteria
- Follow patterns in existing router/service/schema files
- Never modify database migrations directly
- Use type hints on all function signatures
```

For each task, structure your Cursor chat prompt the same way:

```
Goal: [one sentence]
AC: [numbered list]
Scope: [files to change, files NOT to change]
Plan: [ordered steps]
```

Cursor does not enforce this structure — you enforce it by habit.

Scope Boundaries → @file Mentions

Cursor's `@file` syntax controls context. Use it as a constraint fence:

```
@src/orders/router.py @src/orders/service.py
Implement order cancellation. Follow the patterns in these files.
Only change files in src/orders/ and tests/orders/.
```

Mentioning specific files focuses the model's attention and reduces the chance of it inventing patterns that don't match your codebase.

Knowledge Base → .cursorrules + Project Docs

Cursor reads `.cursorrules` on every prompt. Use it for accumulated knowledge:

```
# Known Issues
- bcrypt fails on Python 3.14 — use argon2-cffi
- SQLAlchemy async: no lazy loading, use selectinload()
- JWT refresh: must use DB-level locking to prevent race conditions
```

For larger knowledge bases, maintain a `docs/knowledge/` directory and reference it:

```
@docs/knowledge/dead-ends.md — Read this before implementing auth-related features.
```

Dead Ends → Comment Files or Inline Docs

Cursor does not have a built-in knowledge system. Two options:

Option A: Knowledge files (recommended) — maintain `docs/knowledge/dead-ends.md` and reference with `@`:

```
# Dead Ends

- 2026-03-15: bcrypt C extension fails on Python 3.14 → use argon2-cffi
- 2026-03-18: Celery beat + async → use APScheduler instead
- 2026-03-20: pytest-asyncio auto mode breaks fixtures → use mode=strict
```

Option B: Inline comments — add dead end notes near the relevant code:

```
# DEAD END: Do not use bcrypt here — C extension fails on Python 3.14.
# Use argon2-cffi via get_password_hash(). See docs/knowledge/dead-ends.md.
from auth.utils import get_password_hash
```

QG-0 → Manual (Write AC Before Prompting)

Cursor has no built-in gate. You are the gate. The discipline:

1. Write goal + AC in the chat prompt **before** asking for code
2. If you catch yourself typing "implement X" without AC, stop. Write the AC first.
3. Keep a task template (see Starter Kit below) and paste it every time.

QG-2 → Cursor Review Features + Terminal

After the AI generates code:

1. Use Cursor's diff view to review changes
2. Run tests in the terminal: `pytest tests/ -v`
3. Run type checker: `mypy src/ --strict`
4. Walk through AC one by one — does each criterion have a passing test?

SENAR with GitHub Copilot

GitHub Copilot works primarily through inline completions and Copilot Chat (VS Code sidebar or PR reviews). It has the least structured context control of the three tools.

Goal + AC → Structured Comments Before Code

Copilot reads surrounding code and comments. Use structured comments as the prompt:

```
# TASK: Implement order cancellation endpoint
# GOAL: Allow users to cancel pending orders
# AC-1: POST /orders/{id}/cancel + pending order → 200,
status=cancelled
# AC-2: POST /orders/{id}/cancel + shipped order → 409
# AC-3: POST /orders/{id}/cancel + other user's order → 403
# AC-4: Cancelled order → 409 on re-cancel
# AC-5: Non-existent order → 404
# SCOPE: src/orders/router.py, src/orders/service.py, tests/orders/

@router.post("/orders/{order_id}/cancel")
async def cancel_order(  # Copilot completes from here
```

This is more friction than Claude Code or Cursor, but it works: Copilot uses the comments as context for completions.

Scope Boundaries → File-Level Context

Copilot's context is primarily the open file and related files. Control scope by:

- Having the right files open in tabs (Copilot reads open files)
- Keeping unrelated files closed
- Using Copilot Chat with explicit file references: "Following the pattern in `src/orders/router.py`, implement..."

Knowledge Base → Codebase Documentation (Rule 9.11)

Copilot learns from your codebase. Rule 9.11 (Documentation Completeness) directly improves Copilot output:

- Docstrings on every public function → Copilot generates consistent new functions
- Type hints everywhere → Copilot generates typed code
- `README.md` per module → Copilot understands module purpose

```
"""Order service — handles order lifecycle operations.

Patterns:
- All mutations go through service layer, never router
- Use get_or_404() for entity lookup
- Status transitions validated via Order.can_transition_to()
"""
```

Dead Ends → ADR Files or Doc Comments

Use Architecture Decision Records in `docs/adr/`:

```
# ADR-007: Password Hashing Algorithm

## Status: Accepted

## Context
Need password hashing for auth module. bcrypt C extension fails on
Python 3.14.

## Decision
Use argon2-cffi with argon2id algorithm via shared
get_password_hash() utility.

## Consequences
- All password operations use one utility function
- argon2-cffi must remain in dependencies
- Do not add bcrypt as alternative
```

Copilot reads ADR files when they are open, and Copilot Chat can be directed to them explicitly.

QG-0 → Manual Discipline (Pre-Implementation)

Manual discipline — write goal + AC in a task tracking file or issue before starting AI work. Copilot does not have a built-in pre-implementation gate; use TASK_TEMPLATE.md or issue templates as the enforcement mechanism.

1. Write goal + AC in a GitHub issue or `TASK_TEMPLATE.md` **before** opening the editor
2. If you catch yourself prompting Copilot without AC, stop. Write the AC first.
3. Keep a task template (see Starter Kit below) and fill it in every time.

QG-2 → PR Template + CI Pipeline + PR Review

Since Copilot integrates with GitHub, use PR templates as the post-implementation gate:

```
<!-- .github/PULL_REQUEST_TEMPLATE.md -->
## Task
- [ ] Goal defined
- [ ] Acceptance criteria listed below
- [ ] Scope boundaries stated

## Acceptance Criteria
1.
2.
3.

## Verification
- [ ] Each AC has a passing test
- [ ] mypy/tsc clean
- [ ] No new linting violations

## Knowledge
- Dead ends encountered:
- Decisions made:
```

The PR cannot be opened without filling this in. Reviewers verify against it.

Additionally, Copilot's natural QG-2 includes the CI pipeline:

1. GitHub Actions runs tests, type checker, linter
2. Copilot Chat (PR review mode) reviews the diff
3. Human reviewer checks AC-to-test mapping

This is less immediate than Claude Code's in-session verification, but it works when combined with the PR template.

The Core Knowledge Base (No Infrastructure Required)

You do not need a database, a wiki, or a specialized tool. You need three files in your repository:

```
docs/
  knowledge/
    dead-ends.md      — one line per dead end
    decisions.md      — one paragraph per decision
    gotchas.md        — one line per gotcha
```

dead-ends.md

```
# Dead Ends

Approaches that failed. Check here before trying something new.

- 2026-03-15: bcrypt — C extension fails on Python 3.14. Use argon2-cffi.
- 2026-03-18: Celery beat with async workers — event loop conflicts. Use APScheduler.
- 2026-03-20: pytest-asyncio auto mode — breaks fixtures with scope=session. Use mode=strict.
- 2026-03-21: SQLAlchemy lazy loading in async — not supported. Use selectinload().
- 2026-03-22: Redis Sentinel with asyncio — aioredis deprecated. Use redis-py >= 5.0.
```

Format: `date: thing tried — why it failed. What to use instead.`

One line. Ten seconds to write. Saves hours.

decisions.md

```
# Decisions

Architectural and technical decisions with context.

## Password Hashing: argon2id
**Date:** 2026-03-15
**Context:** Need password hashing for auth module.
**Decision:** Use argon2-cffi with argon2id via `get_password_hash()`
in `src/auth/utils.py`.
**Reason:** OWASP recommendation, Python 3.14 compatible, bcrypt
fails (see dead-ends.md).

## Task Queue: APScheduler over Celery
**Date:** 2026-03-18
**Context:** Need periodic tasks (token cleanup, report generation).
**Decision:** Use APScheduler with AsyncIOScheduler.
**Reason:** Celery beat has event loop conflicts with async FastAPI.
APScheduler runs in-process.
```

Format: title, date, context (one sentence), decision (one sentence), reason (one sentence).

gotchas.md

```
# Gotchas

Things that are technically correct but surprising. Not bugs — just traps.

- SQLAlchemy: `session.refresh()` after commit or you get stale data in tests
- FastAPI: `Depends()` in test overrides must match the exact function, not just signature
- pytest: fixture scope=session + async requires `event_loop` fixture override
- Docker: `COPY requirements.txt .` before `COPY . .` — layer cache invalidation order matters
- PostgreSQL: `SERIAL` vs `GENERATED ALWAYS AS IDENTITY` — use IDENTITY for new tables
```

Format: one line. Context: behavior. That is all.

Why This Works

AI agents read your repository. When `docs/knowledge/dead-ends.md` is in the repo, every AI prompt has access to it — whether through Claude Code's automatic file reading, Cursor's `@file` mentions, or Copilot's open-tab context.

The knowledge base is version-controlled, searchable, and grows with every session. No external tool required.

SENAR Starter Kit

Copy-paste ready templates for immediate use.

TASK_TEMPLATE.md

```
# Task: [slug]

## Goal
[One sentence: what must be accomplished]

## Acceptance Criteria
1. [First criterion — independently testable]
2. [Second criterion]
3. [Third criterion]

## Requirement Link
Story: [story-slug]
BR: [business requirement ID, if applicable]

## Scope
- Change ONLY: [directories/files]
- Do NOT change: [protected files/directories]
- Follow patterns in: [reference files]

## Plan
1. [First step]
2. [Second step]
3. [Third step]

## Notes
[Context, prior dead ends to avoid, related knowledge entries]
```

.github/PULL_REQUEST_TEMPLATE.md

```
## Summary
[One sentence: what this PR does]

## Acceptance Criteria
- [ ] AC-1: [criterion from task]
- [ ] AC-2: [criterion from task]
- [ ] AC-3: [criterion from task]

## Verification
- [ ] All AC have corresponding tests
- [ ] Tests pass locally
- [ ] Type checker clean (mypy/tsc/pyright)
- [ ] Linter clean
- [ ] No new dependencies added (or justified below)

## Dead Ends Encountered
<!-- Approaches that failed during this task. Move to
docs/knowledge/dead-ends.md -->
- None / [description]

## Decisions Made
<!-- Architectural or technical decisions. Move to
docs/knowledge/decisions.md -->
- None / [description]

## Knowledge Entries
<!-- Gotchas, patterns, or non-obvious behaviors discovered -->
- None / [description]
```

docs/knowledge/README.md

```
# Knowledge Base

This directory contains accumulated project knowledge from SENAR
sessions.

## Files

| File | Purpose | Format |
|------|---------|--------|
| `dead-ends.md` | Approaches that failed | One line per entry |
| `decisions.md` | Architectural/technical decisions | One paragraph
per entry |
| `gotchas.md` | Surprising but correct behaviors | One line per
entry |

## How to Use

**Before starting a task:** Check `dead-ends.md` for the area you're
working in.

**During a task:** When an approach fails, add one line to `dead-
ends.md`.
When you make an architectural choice, add a paragraph to
`decisions.md`.
When you discover something surprising, add a line to `gotchas.md`.

**After a task:** Review what you learned. If any dead ends,
decisions, or
gotchas were captured in the PR template, move them to the
appropriate file.

## For AI Agents

If you are an AI coding agent reading this: check `dead-ends.md`
before
attempting any approach in a domain that has entries. Check
`gotchas.md`
for the libraries and frameworks you are about to use. Check
```

```
`decisions.md`
for architectural context.
```

Knowledge Base Options

SENAR does not prescribe a specific knowledge base tool. Common approaches by team size:

- **Solo/Core**: Markdown files in the project repo (e.g., `docs/knowledge/`)
- **Foundation (1-3 Pairs)**: Project wiki, Notion, or structured YAML/JSON in repo
- **Team (3-10)**: Dedicated tool with search — CouchDB+Meilisearch, Confluence, Linear docs
- **Enterprise (10+)**: Federated knowledge bases per project with cross-project search

The key requirement: knowledge must be searchable by AI agents in future sessions.

Choosing Your Tool

SENAR Practice	Claude Code	Cursor	GitHub Copilot
Goal + AC	Prompt text	Prompt text	Structured comments
Scope boundaries	CLAUDE.md + prompt	.cursorrules + @file	Open tabs + prompt
Knowledge base	Memory + MCP	.cursorrules + docs/	docs/ + ADRs
Dead end capture	Memory entries	Knowledge files	ADR files
QG-0 enforcement	/plan skill	Manual discipline	Task template / issue
QG-2 enforcement	/review skill	Terminal + diff view	PR template + CI pipeline
Session handoff	Memory system	Handoff file	PR description
Context persistence	Automatic (memory)	Manual (@file)	Manual (open files)

The tool does not make the process. The process makes the tool useful. A Supervisor with discipline and a plain text editor will outperform a Supervisor with the best AI tool and no acceptance criteria.

SENAR Guide: Agent Configuration in Practice

This chapter bridges the normative requirements of Standard Section 5 (Agent Instrumentation) with practical implementation. It covers how to write operational scripts, define agent profiles, manage script changes, and apply these concepts across different AI development tools.

Anatomy of an Operational Script

An Operational Script is a structured natural-language instruction. Unlike code, scripts are interpreted by an AI agent — which means they must be unambiguous, self-contained, and testable.

Minimal Script Structure

```
Script: commit
Trigger: /commit command or explicit user request

Preconditions:
- Changes exist in the working tree
- No failing tests

Algorithm:
1. Run git status to see all changes
2. Run git diff to review staged and unstaged changes
3. Analyze changes and draft a commit message:
   - Summarize the nature of changes (feature, fix, refactor...)
   - Keep message under 72 characters for the subject line
4. Stage relevant files (prefer specific files over git add -A)
5. Create the commit
6. Run git status to verify success

Postconditions:
- Commit exists in local repository
- Working tree is clean for committed files

Outputs:
- Commit hash
- Summary of what was committed
```

What Makes a Good Script

1. **Explicit preconditions.** If a script assumes something is true, state it. "Tests pass" is a precondition, not an assumption.

2. **Numbered steps with decision points.** When step 3 depends on the outcome of step 2, say so. "If tests fail, stop and report. If tests pass, continue to step 4."

3. **Defined outputs.** What does the script produce? A commit? A report? A status change? Be specific.

4. **Edge cases documented.** "If no changes exist, report this and exit without creating an empty commit."

5. **No platform-specific calls.** Write "Run the test suite" not "Run `npm test`." The script should work regardless of the project's tech stack.

Anti-Patterns

- **Vague instructions:** "Make sure everything is good" — not verifiable
- **Implicit knowledge:** "Use the standard approach" — which standard?
- **Missing error handling:** No guidance for when steps fail
- **Overcoupled scripts:** One script that does planning, implementation, and review

---

Defining Agent Profiles

An Agent Profile restricts what an AI agent can do. The restriction is the point — it prevents the agent from performing actions outside its designated function.

Generator Profile (Primary Development)

```
Profile: Generator
Purpose: Implement code changes for a specific Task

Scripts available:
- implement: Write code to satisfy Task acceptance criteria
- debug: Investigate and fix failures
- commit: Create version-controlled commits
- test: Run and verify test results

Access:
- Read/write: source code, tests, configuration
- Read: task definitions, knowledge base
- Write: task status updates, knowledge entries
- NO access to: deployment, infrastructure, other projects' code

Constraints:
- Changes must stay within Task scope boundaries
- Commits must be atomic (one logical change)
- Must not modify files outside defined scope
```

Reviewer Profile (Independent Verification)

```
Profile: Reviewer
Purpose: Independently verify code changes against acceptance criteria

Scripts available:
- review: Check code against acceptance criteria and standards
- security-audit: Scan for security issues
- report: Generate review findings

Access:
- Read-only: source code, tests, configuration, task definitions
- Write: review comments, findings
- NO access to: code modification, commits, deployment

Constraints:
- Cannot modify any code being reviewed
- Findings must reference specific acceptance criteria
- Must check for AI-specific issues (hallucinated APIs, self-validating
tests)
```

Minimum Viable Profiles

At SENAR Core level, you need at minimum:

1. **Generator** — for development work
2. **Reviewer** — for independent verification

The key insight: even if the same AI agent switches between profiles, the profile switch enforces a cognitive boundary. The Reviewer profile cannot write code, so it evaluates rather than fixes.

Managing Script Changes

Changing a script changes how your AI agent behaves in production. Treat script changes with the same rigor as production code changes.

The Script Change Workflow

```
1. Identify need for change
   ↓
2. Draft the change with rationale
   ↓
3. Review (peer or self, depending on configuration)
   ↓
4. Test on isolated project (Team+)
   ↓
5. Deploy to target project
   ↓
6. Monitor effectiveness (FPSR, error rate)
   ↓
7. Record decision in knowledge base
```

Testing Script Changes

Before deploying a script change across all projects:

1. **Select a representative task** — pick a task similar to what the script will handle
2. **Run the task with the old script** — record FPSR and any issues
3. **Run a similar task with the new script** — compare results
4. **Check for regressions** — did the change break any existing behavior?

Rollback

Every script change should be reversible:

- Scripts are in version control — `git revert` works
- Maintain a version identifier in each script (date or semver)
- Document what to watch for after rollback (tasks in progress may be affected)

Tool-Specific Examples

Claude Code

Claude Code uses `CLAUDE.md` for the Behavioral Contract and `.claude/skills/` for Operational Scripts.

Behavioral Contract (`CLAUDE.md`):

```
# Project Rules
- NEVER commit without explicit permission
- NEVER modify files outside src/ and tests/
- Run tests before every commit
- Ask before installing new dependencies
```

Operational Script (`.claude/skills/review.md`):

```
# Review Skill
Trigger: /review command

Steps:
1. Read the current git diff
2. For each changed file, check:
   a. Does the change match the stated Task goal?
   b. Are there any hardcoded values that should be configurable?
   c. Are error cases handled?
   d. Do tests cover the acceptance criteria, not just code paths?
3. Check for AI-specific issues:
   a. Hallucinated APIs or methods
   b. Non-existent package references
   c. Self-validating test patterns
4. Report findings with specific line references
```

Profile switching: Claude Code does not natively enforce profile permissions. Implement via CLAUDE.md rules: "When performing a review, you are in Reviewer mode. Do NOT modify any code. Only report findings."

OpenAI ChatGPT / GPT-4

Behavioral Contract (system prompt or custom instructions):

```
You are working in Generator mode. You may:
- Read and write source code and tests
- Create commits with descriptive messages
- Search the codebase

You may NOT:
- Modify deployment configuration
- Access external APIs
- Create files outside the project directory
```

Operational Scripts (structured prompts or saved workflows): ChatGPT / GPT-4 does not have a native skill/script mechanism. Scripts are implemented as structured prompt templates that the Supervisor provides at the start of each task.

Cursor

Behavioral Contract (`.cursorrules`):

```
Rules:
- Always run type checking before suggesting changes as complete
- Never modify files not mentioned in the current task
- Prefer editing existing files over creating new ones
```

Operational Scripts (`.cursor/prompts/` or manual invocation): Cursor supports custom prompts that function similarly to operational scripts. Store these in version control alongside the project.

Practical Workflow: Script Change Lifecycle

Scenario: Your "implement" script produces code that frequently has type errors. You want to add a "run type checking" step before marking the implementation as complete.

Step 1: Document the Problem

In your knowledge base:

```
Type: observation
Title: Implementation script produces type errors
Body: In the last 5 tasks, 3 had type checking failures caught at CI
  (QG-2). Adding a type check step to the implement script should
  catch these earlier and improve FPSR.
```

Step 2: Draft the Change

Add to the implement script, after the "write code" step:

```
5. Run the project's type checker
6. If type errors exist:
   a. Fix them
   b. Re-run type checker
   c. Repeat until clean
7. Continue to tests
```

Step 3: Review

For Core: self-review — does the addition make sense? Does it conflict with other steps?

For Team+: peer review — another Supervisor reviews the script change. Check: is the new step clear? Are edge cases handled (what if the project has no type checker)?

Step 4: Test (Team+)

Run one task with the new script on a non-critical project. Did the type check step execute correctly? Did it catch type errors before CI? Did it add unreasonable time?

Step 5: Deploy

Commit the script change. Update the version identifier.

Step 6: Monitor

Over the next 5–10 tasks, track:

- Did type checking failures at CI (QG-2) decrease?
- Did task completion time change significantly?
- Any unexpected side effects?

Step 7: Record

```
Type: decision
Title: Added type checking step to implement script
Body: After 3/5 tasks had type errors at CI, added explicit type check
  step to implement script. First 5 tasks after change: 0/5 type
  errors at CI. ~2 min added per task, saves ~10 min rework.
```

Summary

Concept	Core	Team+
Agent Profiles	Generator + Reviewer (SHOULD)	All 5 with permission enforcement (SHALL)
Operational Scripts	In version control, self-reviewed	Reviewed, tested, registry maintained
Script Changes	Treated as process changes	Tested on isolated project first
Rollback	Version control provides	Registry + explicit rollback procedure
Audit Trail	Knowledge entries	Formal change log

SENAR Reference: Complete Glossary

Alphabetical listing of all SENAR terms. Core terms are defined normatively in SENAR Standard Section 3.

Term	Definition
AI Agent	Software system powered by LLM that generates engineering artifacts under human direction
AI Model Provider	External service providing AI Agent capabilities (e.g., Anthropic, OpenAI, Google); de facto supplier (Standard 3.25)
AI Model Version	Specific version/generation of an AI model; metric baselines are version-dependent (Standard 3.26)
Checkpoint	Context preservation action during a Session
Context	Information provided to AI Agent: goal, AC, constraints, knowledge, traceability
Context Architect	Responsibility: designs requirements as structured AI input, manages traceability
Cost per Task	Metric: total cost / tasks done, segmented by complexity
Cost Predictability	Metric: actual cost / planned cost for an Increment
Cycle Time	Time from Task start to completion (started_at → completed_at)
Dead End	Documented failed approach with reason for abandonment
Defect Escape Rate	Metric: % defects found after Task marked done
Delivery Review	Ceremony: demonstrate software to stakeholders
Exploration	Time-bounded investigation without full Task formality
Federation	Coordination mechanism for multiple Supervisor+AI Pairs
Federation Sync	Ceremony: coordinate multiple Pairs on dependencies
First-Pass Success Rate	Metric: % Tasks completed correctly in one cycle
Flow Manager	Responsibility: session rhythm, cost tracking, flow metrics
Gate Bypass	Documented exception to proceed past a Quality Gate
Increment	Scope-bounded batch of work with objectives and budget
Increment Planning	Ceremony: define objectives, tasks, budget, risks
Increment Retrospective	Ceremony: quantitative review of Increment metrics

Knowledge Capture Rate	Metric: knowledge entries / tasks done
Knowledge Engineer	Responsibility: capture, curate, maintain knowledge base
Knowledge Entry	Documented decision, pattern, gotcha, or dead end
Lead Time	Time from Task creation to completion
Manual Intervention Rate	Metric: % Tasks with manual code writing
Maturity Level	Organization's SENAR adoption depth (L1–L5)
Quality at Input	Principle: defects in requirements cascade to all downstream artifacts; quality is built at input, not checked at output
Quality Gate	Automated enforcement point blocking work unless criteria met
Quality Sweep	Ceremony: periodic comprehensive codebase/KB audit
Requirement	Documented, verifiable statement of need at BR, SR, or TR level (Standard 3.17)
Requirement — Business (BR)	Stakeholder-level need in business terms; source of all downstream requirements (Standard 3.18)
Requirement — System (SR)	System-level capability derived from BR; what the system must do (Standard 3.19)
Requirement — Task (TR)	Implementation-level requirement = Task goal + acceptance criteria (Standard 3.20)
Requirement Hierarchy	Decomposition chain: BR → SR → TR; depth scales by complexity (Standard 3.21)
Requirement Library	Managed repository of verified, reusable requirements stored in Knowledge Base (Team+)
Requirement Pattern	Reusable AC template for common task types (CRUD, migration, UI, integration)
Session	Time-bounded period of supervised AI work
Session End	Ceremony: capture metrics, write handoff, record knowledge
Session Start	Ceremony: load context, select tasks, verify environment
Story	Intermediate grouping of Tasks as stakeholder-visible deliverable
Supervisor	Human who directs AI agents, verifies output, enforces gates

Supervisor+AI Pair	Fundamental production unit: one Supervisor + AI Agent(s)
Task	Atomic unit of tracked work with goal and acceptance criteria
Throughput	Metric: tasks completed per Session
Traceability	Bidirectional linking: every TR → BR upward; every BR → TR(s) downward (Standard 3.24)
Value Stream	End-to-end flow from client request to delivered software
Verification Engineer	Responsibility: audits AI output for correctness and security
Work Type	Functional category of a Task (dev, arch, QA, docs)
WSJF	Prioritization: Cost of Delay / Job Size (from SAFe)

SENAR Reference: Scaling Ratios

These ratios are guidelines. Organizations should adjust based on their domain, AI tooling capabilities, and team maturity.

Responsibility Allocation by Team Size

Pairs	Supervisor	Context Architect	Knowledge Engineer	Flow Manager	Verification Engineer
1–2	1 (covers all)	— (absorbed)	— (absorbed)	— (absorbed)	— (absorbed)
3–5	3–5	1 (covers CA+KE)	(covered by CA)	1 (covers FM+VE)	(covered by FM)
6–10	6–10	1 dedicated	1 dedicated	1 dedicated	1 dedicated
10–20	10–20	2 (by domain)	1	2	2
20–50	20–50	3–5	1–2	2–3 (Federation Coordinators)	3–5
50+	50+	5–10	2–3	5+ (incl. Portfolio Manager)	5–10

Enterprise Roles

Role	Ratio	When Needed
Portfolio Manager	1 per 20–50 Pairs	Multiple value streams or budget oversight
Chief Supervisor	1 per organization	Architectural governance, QG standards
Federation Coordinator	1 per 10–20 Pairs	Cross-team dependency management
Compliance Officer	1 per organization	Regulated industries (ISO, PCI, HIPAA)

Quality Sweep Staffing

Pairs	VE Coverage
1–2	Supervisor self-audits
3–5	1 VE, sweeps rotate across Pairs
6–10	1 dedicated VE, full sweep each cycle
10+	1 VE per 5–10 Pairs, coordinated sweep schedule

SENAR Reference: Efficiency Model

This document provides frameworks for evaluating the efficiency of SENAR adoption. All models use **ratios and multipliers**, not absolute monetary values — organizations substitute their own numbers in any currency.

A. Efficiency Dimensions

SENAR efficiency is measured across four dimensions:

Dimension	What It Measures	Key Ratio
Throughput	Output per production unit	Tasks per Supervisor+AI Pair vs tasks per traditional developer
Quality	Defect prevention and detection cost	Defect cost ratio: early detection vs late detection
Knowledge	Organizational learning retention	Knowledge reuse rate: entries that prevent repeated mistakes
Overhead	Process cost as fraction of delivery	Ceremony + gate time as % of productive session time

---

B. Throughput Model

B.1 Production Unit Comparison

Model	Production Unit	Typical Composition
Traditional	Development Team	5–9 engineers + QA + PM
SENAR	Supervisor+AI Pair	1 Supervisor + AI Agent(s)

Throughput multiplier:

```
T_multiplier = Tasks_per_Pair_per_period / Tasks_per_Developer_per_period
```

Organizations SHOULD measure this ratio during pilot to establish their baseline. The ratio varies significantly by: domain complexity, AI tool capability, Supervisor experience, and context quality.

B.2 Scaling Efficiency

Traditional scaling: adding developers has diminishing returns (Brooks's Law — communication overhead grows as n^2).

SENAR scaling: adding Supervisor+AI Pairs has near-linear returns up to the federation coordination limit, because Pairs operate independently with programmatic dependency tracking.

```
Traditional: Effective_capacity = n × Developer_output × (1 - communication_overhead(n))
SENAR:       Effective_capacity = n × Pair_output × (1 - federation_overhead(n))
```

Where `federation_overhead(n)` grows slower than `communication_overhead(n)` because dependencies are tracked programmatically, not through meetings.

C. Quality Efficiency

C.1 Defect Detection Cost Ratio

Defects caught earlier cost less to fix. This ratio is consistent across organizations:

Detection Point	Relative Cost
During AI generation (same session)	**1×** (baseline)
During Quality Sweep (periodic audit)	**3–5×**
During acceptance testing	**5–10×**
In production	**10–50×**

Quality Gate ROI:

```
Gate_ROI = (Defects_caught × Avg_late_detection_cost) /
Gate_operation_cost
```

Organizations SHOULD measure defect counts at each stage and calculate their own cost ratios.

C.2 First-Pass Efficiency

Higher First-Pass Success Rate (FPSR) means less rework:

```
Rework_cost = Tasks_total × (1 - FPSR) × Avg_rework_cost_per_task
Efficiency_gain = Rework_cost_before_SENAR - Rework_cost_after_SENAR
```

FPSR improves as context quality improves (better acceptance criteria, richer knowledge base, documented dead ends).

D. Knowledge Efficiency

D.1 Dead End Reuse

Each documented Dead End prevents future Supervisors from repeating a failed approach.

```
Dead_End_ROI = Documented_dead_ends × Avg_times_would_be_repeated ×
Avg_exploration_cost
```

The reuse rate for well-documented Dead Ends approaches 100% — nearly every documented dead end prevents at least one repeat within the organization.

D.2 Knowledge Accumulation Effect

As the knowledge base grows, context quality improves, which improves FPSR, which reduces rework:

```
Session N:    FPSR = f(KB_size_at_N, Context_quality)
Session N+K:  FPSR' > FPSR  (if KB is maintained and growing)
```

This creates a compound efficiency gain — each Increment is more efficient than the last, up to the plateau where most common patterns and pitfalls are documented.

E. Overhead Model

E.1 Process Overhead Ratio

```
Overhead_ratio = Time_on_ceremonies_and_gates / Total_session_time
```

Target: overhead < 15% of session time for Core/Foundation, < 20% for Team.

Activity	Core/Foundation	Team
Session Start	2–5 min	2–5 min
Session End	5–10 min	5–10 min
Quality Gate checks	Automated (0 min)	Automated (0 min)
Quality Sweep	Periodic (amortized)	Periodic (amortized)
Federation Sync	N/A	5–10 min per sync
Increment Planning	Amortized	1 session per Increment
Retrospective	Amortized	30–60 min per Increment

E.2 Overhead Break-Even

SENAR overhead pays for itself when defect prevention savings exceed process cost:

```
Break_even: Gate_cost + Ceremony_cost < Defects_prevented × Avg_defect_cost
```

Organizations SHOULD calculate this after 3 Increments with measured data.

F. Comparison Framework

F.1 Traditional Team vs SENAR Team

To compare delivery efficiency for the same scope:

Metric	Traditional Team	SENAR Team	How to Measure
Headcount	N developers + QA + PM	M Supervisors + support roles	Count
Throughput	Tasks per period	Tasks per period	Same task granularity
Defect rate	Defects per 100 tasks	Defects per 100 tasks	Same counting method
Lead time	Requirement → production	Requirement → production	Same milestones
Rework rate	% tasks requiring rework	% tasks requiring rework (1 - FPSR)	Same definition
Knowledge retention	Bus factor, onboarding time	KB coverage, onboarding time	Measured

F.2 Decision Criteria

SENAR is more efficient when:

- AI tools can generate the majority of implementation artifacts for the domain
- Throughput multiplier (B.1) exceeds overhead ratio (E.1) — net productivity gain
- Defect prevention savings (C.1) exceed quality gate costs — net quality gain
- Knowledge accumulation (D.2) provides compounding returns over time

SENAR is less efficient when:

- Domain is poorly suited for AI generation (novel research, highly regulated manual processes)
- AI tooling costs exceed the value of throughput gains
- Organization cannot invest in tooling infrastructure (task tracker, CI/CD, knowledge base)
- Team is too small to benefit from process structure (1 developer on a side project)

G. Evaluation Worksheet

Organizations evaluating SENAR adoption SHOULD measure these during a pilot (minimum 3 Increments):

Metric	Pilot Value	Baseline (before SENAR)	Delta
Tasks per Pair per session	___	___ (per developer)	×___
FPSR	___%	N/A (new metric)	—
Defect Escape Rate	___%	___%	___%
Rework rate	___%	___%	___%
Session overhead (min)	___	N/A	—
Knowledge entries created	___	0	+___
Dead Ends documented	___	0	+___

Decision Rule

```
IF throughput_multiplier > 1.0 + overhead_ratio
AND defect_escape_rate <= baseline_defect_rate
THEN SENAR is providing net efficiency gain → consider scaling
```

H. Red Flags

Signs that SENAR adoption is reducing rather than improving efficiency:

Red Flag	What It Means	Action
Overhead ratio > 30%	Process overhead exceeds value	Simplify: reduce to MVS, automate ceremonies
FPSR declining over time	Context quality degrading	Audit knowledge base, review AC quality
Throughput multiplier < 1.0	Pairs slower than traditional developers	Wrong domain for AI, or insufficient Supervisor training
Gate Bypass rate > 20%	Gates don't match reality	Recalibrate gate criteria
Knowledge entries = 0	Knowledge capture abandoned	Reinforce Dead End documentation at minimum
Sessions consistently exceed duration limit	No discipline	Enforce checkpoints, review causes

SENAR Reference: Governance and Compliance Annex

This annex maps SENAR practices to governance, regulatory, and compliance requirements for organizations using AI-native development teams. It is intended for compliance officers, auditors, legal counsel, and engineering leadership evaluating SENAR adoption in regulated environments.

Disclaimer: This document provides recommended practices and compliance mapping guidance. It does not constitute legal advice. Organizations should consult qualified legal counsel for jurisdiction-specific regulatory interpretation.

A. Responsibility Model

A.1 The Accountability Principle

In SENAR, **the Supervisor is accountable for all AI output they approve**. This is not a delegation of responsibility to the AI agent — it is an explicit acceptance of responsibility by the human who directs, reviews, and approves the work.

The accountability chain follows a clear principle:

> ***AI generates. Human approves. Human is responsible.***

This principle is embedded structurally in SENAR through Quality Gates. When a Supervisor passes a Task through QG-2 (Implementation Gate), they are attesting that:

- CI passes, tests pass, types are clean (automated verification);
- Acceptance criteria are satisfied (human judgment);
- No security vulnerabilities were detected (tooling-assisted);
- The output is fit for its intended purpose (professional judgment).

This attestation is the compliance-relevant act. It is recorded, timestamped, and attributable to a specific individual.

A.2 Responsibility by Quality Gate

Gate	Who Is Responsible	What They Attest
QG-0 (Context)	Supervisor / Context Architect	Task is well-defined with verifiable acceptance criteria
QG-1 (Requirements)	Context Architect	Business requirement is approved and properly decomposed
QG-2 (Implementation)	Supervisor	AI output meets acceptance criteria, CI passes, no security issues
QG-3 (Verification)	Verification Engineer / Peer Reviewer	Code reviewed, acceptance tests pass, no regressions
QG-4 (Acceptance)	Stakeholder / Context Architect	Software meets business requirements, ready for release

Each gate passage constitutes a documented approval decision by an identified human.

A.3 Gate Bypass Responsibility

SENAR Standard Section 8.6(c) requires that Gate Bypasses include justification, risk assessment, remediation plan, and senior approval. From a compliance perspective:

- **The person who approves the bypass owns the risk** for any downstream consequences.
- Gate Bypass records SHALL be preserved as audit evidence.
- Organizations SHOULD track Gate Bypass rate (Standard 8.6(d)) as a compliance health metric.
- Auditors should treat elevated Gate Bypass rates as an indicator of process pressure that warrants investigation.

A.4 Escalation Paths

Situation	Escalation Path
Supervisor is uncertain about AI output correctness	Escalate to Verification Engineer or peer Supervisor for independent review
AI output touches high-risk areas (security, auth, payment, data migration)	SHALL trigger peer review per risk-based review policy (Standard 8.7)
Defect found post-release in AI-generated code	Incident response process (Section F); trace to originating Task and Supervisor
Supervisor suspects AI hallucination or fabricated references	Stop, verify independently, document as Dead End if approach is abandoned
Regulatory or compliance implications unclear	Escalate to Compliance Officer (or equivalent organizational role) before proceeding

A.5 Shared Responsibility in Team+ Configurations

In Team+ configurations, responsibility is distributed across dedicated roles:

- **Context Architect** is responsible for requirement quality and traceability completeness.
- **Knowledge Engineer** is responsible for knowledge base accuracy and freshness.
- **Flow Manager** is responsible for process adherence and metric collection.
- **Verification Engineer** is responsible for independent verification and Quality Sweep thoroughness.
- **Supervisor** remains responsible for the specific AI output they direct and approve.

This distribution does not dilute individual accountability — it creates a chain of documented responsibilities, each with its own audit trail.

B. Audit Trail Requirements

B.1 SENAR Artifacts as Audit Evidence

SENAR's structure produces the following artifacts, each of which constitutes audit evidence:

Artifact	What It Contains	Compliance Value
Task record	Goal, acceptance criteria, requirement link, work type, lifecycle timestamps, assigned Supervisor	Demonstrates planned, authorized work with traceability
Quality Gate records	Gate ID, pass/fail, timestamp, approver, automated check results	Demonstrates enforcement of controls at each stage
Gate Bypass records	Justification, risk assessment, remediation plan, approver	Demonstrates controlled exception management
Session logs	Start/end timestamps, tasks worked, checkpoint records, handoff notes	Demonstrates supervised execution with bounded scope
Handoff documents	Session summary, next steps, warnings, open issues	Demonstrates continuity of supervision across sessions
Knowledge entries	Decisions, patterns, Dead Ends, gotchas — timestamped and categorized	Demonstrates organizational learning and rationale capture
Increment Planning records	Objectives, task pool, budget, risk register	Demonstrates planned delivery with risk management
Retrospective records	Metrics review, planned vs actual, improvement actions	Demonstrates continuous improvement and management review
Quality Sweep reports	Audit scope, findings, remediation actions	Demonstrates periodic independent verification
Metrics data	Throughput, Lead Time, FPSR, DER, Cost Predictability, etc.	Demonstrates measurement-based process management

B.2 Quality Gate to Audit Evidence Mapping

Quality Gate	Evidence Produced	Demonstrates
QG-0 (Context)	Task created with goal, AC, requirement link	Work authorization and scope definition
QG-1 (Requirements)	Approved requirement, decomposition record	Requirements management and approval
QG-2 (Implementation)	CI results, test results, Supervisor approval record	Verification of deliverable, change control
QG-3 (Verification)	Review record, acceptance test results, security scan	Independent verification, security assessment
QG-4 (Acceptance)	Stakeholder approval, staging verification, Delivery Review record	Release authorization, acceptance testing

B.3 Demonstrating Adequate Supervision

An auditor evaluating whether AI-generated code was adequately supervised should examine:

1. **Task definition quality.** Was QG-0 enforced? Did the Task have clear, verifiable acceptance criteria before AI work began? A well-defined Task demonstrates intentional direction, not open-ended AI generation.

2. **Gate passage records.** Did QG-2 pass with documented Supervisor approval? Were automated checks (CI, tests, lint, security scan) executed and recorded? Automated gate records are stronger evidence than manual checklists.

3. **Risk-appropriate review.** Was the risk level correctly classified? Did high-risk changes receive peer review as required by Standard 8.7? Consistent application of risk-based review is a key indicator.

4. **Session discipline.** Were Sessions bounded in duration? Were checkpoints performed at the required cadence? Unbounded sessions without checkpoints suggest inadequate attention.

5. **Knowledge capture.** Were decisions documented? Were Dead Ends recorded? Active knowledge capture indicates reflective supervision, not passive acceptance.

6. **Metrics trend.** Is the Defect Escape Rate stable or improving? A rising DER suggests degrading supervision quality.

7. **Gate Bypass rate.** Are bypasses rare and well-justified? Frequent bypasses indicate systemic process issues.

B.4 Retention Recommendations

Artifact Category	Recommended Minimum Retention	Rationale
Task and gate records	Duration of software lifecycle + regulatory retention period	Core traceability evidence
Session logs and handoffs	3 years or regulatory minimum (whichever is longer)	Supervision evidence
Knowledge entries	Indefinite (active curation)	Ongoing operational value
Metrics data	3 years minimum	Trend analysis and audit evidence
Gate Bypass records	Duration of software lifecycle	Risk acceptance evidence
Incident records	Per regulatory requirement (typically 5-7 years)	Post-incident analysis

C. Regulatory Mapping

C.1 ISO 9001:2015 — Quality Management Systems

ISO 9001 Clause	Requirement Summary	SENAR Artifact	How SENAR Satisfies
6.1 — Actions to address risks and opportunities	Identify risks affecting QMS, plan actions	Increment Planning risk register; Gate Bypass risk assessments; risk-based review classification (Standard 8.7)	Each Increment begins with documented risk identification. Gate Bypasses require explicit risk assessment. Review depth is determined by risk level.
7.5 — Documented information	Create, update, and control documented information	Task records, gate records, knowledge entries, session logs, handoffs	All SENAR artifacts are timestamped, attributable, and retained. Knowledge entries are curated for freshness (Knowledge Engineer responsibility).
8.1 — Operational planning and control	Plan and control processes for product/service provision	Increment Planning with objectives, task pool, and budget; QG-0 enforcing task definition before work	Work is planned at Increment level, authorized at Task level, and controlled through automated Quality Gates.
8.5 — Production and service provision	Controlled conditions for production	Session discipline (bounded duration, checkpoints); Supervisor direction of AI agents; automated CI/CD enforcement	AI production occurs under supervised, time-bounded conditions with automated controls. Manual interventions are tracked.
8.6 — Release of products and services	Verify product/service requirements met before release	QG-4 (Acceptance Gate): stakeholder approval, staging verification, QG-3 still passing	Release requires documented verification and stakeholder acceptance.

9.1 — Monitoring, measurement, analysis, and evaluation	Determine what to monitor and measure	8 defined metrics (4 mandatory, 4 recommended); Increment Retrospective with quantitative review	Metrics are collected automatically, reviewed at each Retrospective, with baselines established over 3+ Increments.
10.1 — Improvement: Nonconformity and corrective action	React to nonconformities, take corrective action	Retrospective improvement actions (specific, measurable, time-bounded, assigned); Quality Sweep findings and remediation	Retrospectives produce assigned corrective actions. Quality Sweeps identify nonconformities. Dead Ends document failed approaches.

C.2 SOC 2 Type II — Trust Services Criteria

SOC 2 Criterion	Requirement Summary	SENAR Artifact	How SENAR Satisfies
CC6.1 — Logical and physical access controls	Restrict access to authorized users and processes	SENAR does not prescribe access control tooling, but: Supervisor role assignment controls who may approve gate passage; Task assignment documents authorized workers; Session logs document who performed what work and when	Organizations must supplement SENAR with infrastructure-level access controls. SENAR provides the process-level authorization records.
CC7.1 — Detection of unauthorized or malicious activity	Monitor and detect anomalies	Quality Sweeps detect scope creep, unauthorized changes, configuration drift; Version control rules (Standard 10.6) mandate atomic commits with secrets detection; DER metric tracks defects escaping gates	Quality Sweeps and automated scanning provide detection. Organizations must supplement with infrastructure monitoring.
CC7.2 — Monitoring of system components	Monitor system components to detect anomalies	Session logs document all AI-directed changes; Gate records document all approvals; Metrics track process health indicators (FPSR, DER)	SENAR provides process-level monitoring. Organizations must supplement with system-level monitoring (APM, logging, alerting).
CC8.1 — Changes to infrastructure, data, software, and procedures	Manage changes through a controlled process	QG-0 through QG-4 constitute a change management pipeline; Task records document change authorization; Gate records document change verification; Gate Bypasses document controlled exceptions	SENAR's Quality Gate pipeline is a change management process. Every change is authorized (Task), verified (QG-2), independently reviewed (QG-3 for Team+), and released (QG-4).

C.3 GDPR — Considerations for AI Tools Processing Personal Data

GDPR applies when AI coding tools process personal data. This can occur when:

- Source code contains personal data (e.g., test fixtures with real user data, hardcoded credentials);
- AI prompts include personal data from production systems (e.g., debugging with real data);
- AI tool providers process and potentially retain prompt data;
- Session logs contain personal data referenced during development.

GDPR Consideration	Recommended SENAR Practice
Lawful basis for processing	Organizations should establish lawful basis (typically legitimate interest or contract performance) for any personal data sent to AI tools. Document this in data processing records.
Data minimization	QG-0 context preparation SHOULD exclude personal data. Use synthetic or anonymized data in AI prompts. Include data classification check in Quality Sweep scope.
Data processing agreements	Organizations processing EU personal data via cloud-hosted AI tools SHALL have Data Processing Agreements (DPAs) in place with AI tool providers covering prompt data processing and retention.
Right to erasure	Session logs containing personal data must be subject to erasure processes. Organizations should design session logging to minimize personal data capture.
Data protection impact assessment	Organizations SHOULD conduct a DPIA before adopting AI coding tools that will process personal data, particularly when using cloud-hosted models.
Cross-border transfers	When AI tools process data outside the EEA, organizations must ensure adequate transfer mechanisms (SCCs, adequacy decisions) are in place.

C.4 General Software Audit Requirements

Audit Requirement	SENAR Artifact	How SENAR Satisfies
Traceability from requirement to implementation	Task → requirement link (QG-0 mandate); Story → Task decomposition; Increment objectives → Stories	Every Task links to its parent requirement. Full traceability chain from business objective to implementation.
Change authorization	Task creation (authorization to work); QG-2 passage (authorization to close); QG-4 passage (authorization to release)	Changes are authorized at three levels: work initiation, implementation approval, and release.
Segregation of duties	Supervisor directs AI; Verification Engineer reviews independently (Team+ configurations); Stakeholder approves release	Team+ configurations provide role separation. Core/Foundation configurations should document compensating controls.
Evidence of testing	QG-2 mandates CI pass, tests pass; QG-3 mandates acceptance tests pass; automated test results recorded as gate evidence	Test execution is automated and recorded as part of gate passage.
Incident management	Task-based incident tracking; traceability from incident to originating code change; post-incident review process (Section F)	Incidents are traceable through the full chain: incident → code → Task → requirement → Supervisor.
Management review	Increment Retrospective with quantitative metrics; Quality Sweep reports; Delivery Review records	Regular, structured reviews with documented outcomes and improvement actions.
Continuous improvement	Retrospective improvement actions; DER and FPSR trending; Knowledge base growth	Improvement is measured (metrics), documented (actions), and verified (subsequent Retrospective review).

C.5 EU AI Act (Regulation 2024/1689)

AI coding assistants typically fall under 'limited risk' or 'general-purpose AI' categories. Key obligations:

- **Article 50: Transparency** — users must be informed they interact with AI. SENAR's explicit labeling of AI-generated work satisfies this.
- **Article 53: Obligations for GPAI model providers** — relevant when organizations fine-tune or deploy models.
- Organizations operating in the EU SHOULD review their AI tool usage against the Act's requirements and maintain documentation of compliance measures.

C.6 NIST AI Risk Management Framework (AI RMF 1.0)

SENAR practices align with NIST AI RMF functions:

- **Govern**: Roles (Section 4), Governance (this annex)
- **Map**: Risk-based review tiers (Section 8.7), Data classification (D.1)
- **Measure**: Metrics (Section 9), Quality Sweeps (Section 7.4)
- **Manage**: Quality Gates (Section 8), Incident Response (F.1)

Organizations subject to US federal AI governance requirements SHOULD document this mapping.

C.7 ISO/IEC 27001:2022

SENAR controls map to ISO 27001 Annex A:

- **A.8.25** (Secure development lifecycle): Quality Gates, Verification Checklist
- **A.8.28** (Secure coding): Rules 10.6, 10.15, AI Output Review
- **A.8.9** (Configuration management): Rules 10.13, 10.14
- **A.5.12** (Classification of information): Data classification (D.1)
- **A.8.15** (Logging): Audit trail requirements (B.1)

Organizations seeking ISO 27001 certification SHOULD map SENAR artifacts to their Statement of Applicability.

D. Data Governance for AI Tools

D.1 Data Classification for AI Interactions

Before sending any data to an AI coding tool, organizations should classify that data and apply appropriate controls:

Classification	Definition	AI Tool Policy	Examples
Public	Information intended for public disclosure	Any AI tool (cloud or on-premise)	Open-source code, public documentation, published APIs
Internal	Information for internal use, low sensitivity	Cloud AI tools with DPA; on-premise AI tools	Internal architecture docs, non-sensitive business logic, internal tooling code
Confidential	Sensitive business information	On-premise AI tools preferred; cloud only with explicit DPA, encryption, and no-retention guarantees	Proprietary algorithms, customer-facing feature code, competitive differentiators
Restricted	Highest sensitivity; regulatory or contractual obligations	On-premise AI tools only; no cloud transmission	Personal data (GDPR), payment card data (PCI DSS), health records (HIPAA), authentication secrets, encryption keys

D.2 AI Tool Selection Criteria

Organizations should evaluate AI coding tools against the following criteria, weighted by their data classification requirements:

Criterion	Questions to Answer
Data retention	Does the provider retain prompts or generated output? For how long? Can retention be disabled?
Training opt-out	Does the provider use customer data for model training? Can this be contractually excluded?
Data residency	Where is data processed and stored? Does this comply with applicable data residency requirements?
Encryption	Is data encrypted in transit (TLS 1.2+) and at rest? Who holds the encryption keys?
Access controls	Who at the provider can access customer data? Under what circumstances?
Audit rights	Does the contract include audit rights or third-party audit reports (SOC 2, ISO 27001)?
Subprocessors	Does the provider use subprocessors? Are they disclosed and contractually bound?
Incident notification	What are the provider's breach notification obligations and timelines?

D.3 On-Premise vs. Cloud AI: Decision Framework

Factor	Cloud AI Appropriate	On-Premise AI Required
Data classification	Public or Internal	Confidential or Restricted
Regulatory environment	No data residency requirements	Data sovereignty or residency mandates
Contractual obligations	No customer restrictions on cloud processing	Customer contracts prohibit cloud AI processing
Risk tolerance	Organization accepts cloud provider risk	Zero tolerance for external data exposure
Cost-performance trade-off	Cloud cost is acceptable for capability	On-premise cost justified by compliance requirements

Organizations operating in regulated industries (finance, healthcare, government, defense) should default to on-premise AI tools for all non-public data unless a specific risk assessment justifies cloud use.

D.4 Data Retention for AI Interactions

Data Type	Retention Recommendation	Notes
Session logs (sanitized)	Per audit trail retention policy (Section B.4)	Remove personal data before long-term retention
Raw AI prompts	Minimal retention; purge after session close unless regulatory requirement	Prompts may contain sensitive context
AI-generated output	Retained as part of version-controlled source code	Standard code retention policies apply
Task and gate records	Duration of software lifecycle + regulatory period	Core compliance artifacts
AI tool usage metrics	3 years minimum	Cost tracking, audit evidence

D.5 Right to Deletion

A critical question for organizations subject to GDPR or similar privacy regulations: if personal data was included in an AI prompt, can its influence be removed from the model?

Current reality: For cloud-hosted AI models, organizations generally cannot guarantee removal of training influence from model weights, even if the provider deletes stored prompt data. This is a fundamental limitation of current large language model architectures.

Recommended practices:

1. **Prevent rather than remediate.** Data classification and QG-0 context preparation should exclude personal data from AI prompts. Prevention is more reliable than post-hoc deletion.
2. **Contractual no-training clauses.** Ensure AI tool contracts explicitly exclude customer data from model training.
3. **Prompt data deletion.** Ensure contracts include prompt data deletion obligations and verify compliance.
4. **On-premise models for restricted data.** When deletion guarantees are required, use on-premise models where the organization controls the full data lifecycle.
5. **Document the limitation.** If an organization cannot guarantee deletion of training influence, document this as a known risk in the DPIA and apply compensating controls (data minimization, synthetic data).

E. Intellectual Property and Licensing

Note: AI-generated code IP is an evolving legal landscape with significant jurisdictional variation. The following represents recommended practices as of the time of writing. Organizations should obtain qualified legal counsel for their specific jurisdiction.

E.1 Ownership of AI-Generated Code

The legal status of AI-generated code ownership varies by jurisdiction and remains unsettled in many:

Jurisdiction Approach	Current Position	Implication for SENAR
Human authorship required	Several jurisdictions require human creative contribution for copyright. Purely machine-generated output may not be copyrightable.	SENAR's Supervisor model strengthens IP claims: the Supervisor provides creative direction (Task goals, acceptance criteria, architectural decisions) and exercises judgment in selection and approval.
Work-for-hire / employment	In many jurisdictions, employer owns work created by employees in course of employment.	AI tool output directed by employee Supervisors is likely covered under existing employment IP agreements, but organizations should verify.
AI tool provider terms	Most AI tool providers assign output rights to the user, but terms vary.	Organizations should review AI tool terms of service for IP assignment clauses and ensure they are compatible with organizational requirements.

Recommended practices:

1. **Update employment and contractor agreements** to explicitly address AI-assisted and AI-generated code.
2. **Review AI tool provider terms** to confirm output ownership assignment.
3. **Document human creative contribution** through SENAR's Task records (goal, acceptance criteria, architectural decisions, Supervisor review notes). This documentation strengthens authorship claims.
4. **Maintain records of Supervisor direction** — the context provided to the AI, the selection and modification of AI output, and the judgment applied during review.

E.2 License Contamination Risk

AI models trained on open-source code may generate output that reproduces or closely resembles code under copyleft licenses (GPL, AGPL, LGPL). If such output is incorporated into proprietary software without compliance, this creates license contamination risk.

Risk factors:

- AI models trained on public code repositories without license filtering;
- Generated code that closely matches existing open-source implementations;
- Insufficient review of generated code for license-encumbered patterns.

Recommended practices:

1. **Automated license scanning.** Include license scanning tools in the CI pipeline (enforced at QG-2). Scan both direct dependencies and generated code for known license-encumbered patterns.
2. **Dependency audit at QG-3.** For Team+ configurations, include dependency license audit in the QG-3 verification scope.
3. **Quality Sweep coverage.** Include license compliance in Quality Sweep scope (dependency health audit).
4. **AI tool selection.** Evaluate AI tool providers' training data policies and any indemnification they offer against IP claims.
5. **Code provenance documentation.** For high-value or high-risk code, document whether it was AI-generated, human-written, or a combination.

E.3 Documentation Requirements for IP Compliance

Organizations seeking to demonstrate IP compliance for AI-generated code should maintain:

Document	Purpose	SENAR Source
AI tool inventory	List of AI tools used, their terms of service, IP provisions	Organizational policy (outside SENAR scope)
Task records with Supervisor attribution	Document human creative direction and judgment	Task records, gate records, session logs
License scan results	Demonstrate absence of license contamination	QG-2 and QG-3 automated scan results
Dependency audit records	Document license compliance for all dependencies	Quality Sweep reports
Human contribution records	Strengthen authorship claims	Context preparation records, review notes, knowledge entries documenting design decisions

---

F. Incident Response for AI-Generated Defects

F.1 Response Process

When a production incident is traced to AI-generated code, the following process applies in addition to the organization's standard incident response:

1. **Triage and contain.** Standard incident response: assess severity, contain impact, restore service.
2. **Trace the origin.** Use SENAR's traceability chain:
    - Production incident → code change (version control);
    - Code change → Task record (commit references Task);
    - Task record → requirement (QG-0 requirement link);
    - Task record → Supervisor (Task assignment and QG-2 approval);
    - Task record → Session (session log with timestamps and context).
3. **Assess gate effectiveness.** For each Quality Gate the defective code passed through:
    - Did the gate execute? (Check gate records.)

    - Did automated checks pass? (Were checks adequate for this defect type?)
    - Did human review occur? (Was it risk-appropriate per Standard 8.7?)
    - Was a Gate Bypass involved? (If so, was the bypass risk assessment accurate?)
4. **Classify the root cause.** Use the AI-specific root cause taxonomy (Section F.2).
5. **Remediate.** Fix the immediate defect. Create a Task for any systemic improvement.
6. **Credential rotation.** If credentials or secrets are exposed (committed to VCS, included in AI context, or logged), the organization SHALL rotate affected credentials within 24 hours and audit access logs for unauthorized use.
7. **Post-incident review.** Conduct a structured review addressing:
    - Was supervision adequate for the risk level of this change?
    - Were Quality Gates effective? Should gate criteria be strengthened?
    - Was the AI agent's context sufficient? Should knowledge entries be created?
    - Should the risk classification for this type of change be elevated?

Organizations SHOULD define severity-based response timeframes for AI-generated vulnerabilities. Recommended: Critical — 4 hours, High — 24 hours, Medium — 72 hours, Low — next increment.

F.2 AI-Specific Root Cause Taxonomy

Standard root cause categories (logic error, integration error, performance issue) apply to AI-generated code. The following additional categories are specific to AI-native development:

Root Cause Category	Description	Indicator	Preventive Control
Hallucination	AI fabricated an API, library, or behavior that does not exist	Code references nonexistent functions, packages, or configurations	QG-2 automated checks (compilation, tests); Supervisor verification of external references
Context gap	AI lacked necessary context to produce correct output	Code contradicts existing architecture, duplicates existing functionality, or violates undocumented constraints	Improved QG-0 context preparation; knowledge base entries for constraints and conventions
Stale context	AI used outdated information (deprecated API, changed requirement)	Code uses deprecated patterns or APIs; behavior matches old requirements	Knowledge base freshness maintenance (Knowledge Engineer); Session Start context refresh
Gate bypass	Quality Gate was bypassed and the defect was in bypassed scope	Gate Bypass record exists for the relevant gate	Review Gate Bypass approval process; reduce bypass rate
Insufficient gate criteria	Quality Gate passed but criteria were inadequate to detect the defect	Gate records show passage; defect type was not covered by automated checks	Strengthen gate criteria; add test coverage for defect type
Scope creep	AI generated changes beyond the Task scope that introduced the defect	Commit includes changes unrelated to Task acceptance criteria	QG-2 scope review (Standard 10.6(c)); atomic commit enforcement
Supervision gap	Supervisor approved without adequate review	No evidence of substantive review; rubber-stamp pattern in gate records	Session duration limits; review quality metrics; Verification Engineer audits
Accumulation effect	Multiple individually-correct AI changes interact	No single commit is defective; issue emerges from combination	Quality Sweeps (architectural conformance check);

	to create a systemic issue		integration testing at QG-3

F.3 Traceability Chain

The complete traceability chain for incident investigation:

```
Production Incident
  └─► Code Change (commit hash, diff, timestamp)
        └─► Task Record (goal, AC, work type, Supervisor)
              ├─► Requirement (Story / business requirement)
              ├─► QG-0 Record (context quality at task start)
              ├─► QG-2 Record (implementation verification)
              │     ├─► Automated check results (CI, tests, lint,
security)
              │     └─► Supervisor approval (who, when)
              ├─► QG-3 Record (if applicable: review, acceptance tests)
              ├─► Session Log (when, duration, checkpoints, context)
              └─► Knowledge Entries (decisions, Dead Ends, patterns)
```

This chain enables auditors and incident investigators to reconstruct the full decision history from business requirement through implementation to production deployment.

G. What SENAR Does Not Cover

SENAR is a process methodology for supervised AI-native development. The following compliance-relevant areas are outside SENAR's scope and require additional organizational controls:

Area	Why It Is Outside SENAR Scope	What Organizations Need
Infrastructure access controls	SENAR prescribes process roles, not system permissions	IAM policies, RBAC, MFA, privileged access management
Network security	SENAR is tool-agnostic; network architecture is implementation-specific	Firewalls, VPNs, network segmentation, DDoS protection
Data encryption standards	SENAR does not prescribe cryptographic controls	Encryption-at-rest, encryption-in-transit, key management
Business continuity / disaster recovery	SENAR addresses development process, not operational resilience	BCP/DR plans, backup procedures, RTO/RPO targets
Physical security	SENAR is a software development methodology	Facility access, environmental controls
Employee background checks	SENAR defines responsibilities, not HR processes	Background verification, security clearance procedures
Vendor management (beyond AI tools)	SENAR addresses AI tool data governance only	Third-party risk management program
Privacy by design	SENAR supports data minimization in AI prompts but does not replace privacy architecture	Privacy impact assessments, data mapping, consent management
AI model governance	SENAR governs AI usage in development, not AI model training or deployment	Model risk management, bias testing, model monitoring (relevant for organizations that train or fine-tune models)
Regulatory reporting	SENAR produces audit evidence but does not automate regulatory submissions	Reporting workflows, regulatory calendars, submission procedures

Organizations adopting SENAR should integrate it into their broader governance, risk, and compliance (GRC) framework rather than treating it as a standalone compliance solution.

G.1 Compensating Controls for Core/Foundation Configuration

SENAR's Core configuration (1 Pair) and Foundation configuration (1–3 Pairs) combine multiple responsibilities in fewer people. This creates a segregation-of-duties gap that may

concern auditors:

Gap	Compensating Control
Supervisor self-reviews (no independent Verification Engineer)	Automated gate enforcement (QG-2 automated checks are independent of the Supervisor); periodic Quality Sweeps; peer review for high-risk changes
Supervisor absorbs Context Architect role	QG-0 enforcement ensures minimum task definition quality regardless of who fills the role
No dedicated Flow Manager	Automated metrics collection reduces dependence on manual oversight; Session duration limits are enforceable through tooling

Organizations in regulated industries operating at Core or Foundation scale should document these compensating controls and assess whether they are adequate for their regulatory obligations. Transitioning to Team configuration may be necessary to satisfy segregation-of-duties requirements.

H. Implementation Checklist

Organizations adopting SENAR in a regulated environment should address the following:

- [ ] Establish data classification policy for AI tool interactions (Section D.1)
- [ ] Evaluate and select AI tools based on data classification and compliance requirements (Section D.2)
- [ ] Execute data processing agreements with cloud AI tool providers (Section D.3)
- [ ] Update employment and contractor IP agreements for AI-generated code (Section E.1)
- [ ] Integrate license scanning into CI pipeline at QG-2 (Section E.2)
- [ ] Configure Quality Gate automation to produce audit-grade records (Section B.1)
- [ ] Establish Gate Bypass approval process with documentation requirements (Section A.3)
- [ ] Define data retention policies for SENAR artifacts (Section B.4)
- [ ] Conduct DPIA if AI tools will process personal data (Section C.3)
- [ ] Document compensating controls if operating at Core/Foundation configuration (Section G.1)
- [ ] Establish AI-specific root cause categories in incident management process (Section F.2)
- [ ] Include AI tool compliance in Quality Sweep scope (Section D, E)
- [ ] Train Supervisors on data classification and AI-specific compliance obligations

- [ ] Integrate SENAR audit trail into existing GRC tooling and reporting

---

SENAR Reference: Tooling Requirements

This document specifies what tooling must support for each SENAR component. Requirements are split into two tiers:

- **Core/Foundation** — minimum for SENAR Core (1 Pair) and Foundation (1-3 Pairs) configurations, Maturity Level 2
- **Team+** — full requirements for SENAR Team and Enterprise configurations (3+ Pairs, Maturity Level 3+)

SENAR is tool-agnostic by design. These are capability requirements, not product recommendations.

---

1. Task Tracker

The task tracker is the system of record for all units of work. Every Task, Story, and Increment lives here.

1.1 Required Fields

Field	Type	Core/Foundation	Team+	Notes
`goal`	Text	SHALL	SHALL	What the Task must accomplish, stated as an outcome
`acceptance_criteria`	Text	SHALL	SHALL	Verifiable conditions for Task completion
`status`	Enum	SHALL	SHALL	Minimum states: `planning`, `active`, `done`, `blocked`
`work_type`	Enum	SHALL	SHALL	Values: `dev`, `arch`, `qa`, `docs`, `infra`, `ops`
`complexity`	Enum	SHOULD	SHALL	Values: `trivial`, `simple`, `moderate`, `complex`
`story_id`	Reference	SHOULD	SHALL	Link to parent Story or requirement
`requirement_link`	Reference	SHOULD	SHALL	Traceability link to BR/SR/TR (Standard 3.21)
`requirement_level`	Enum	—	SHALL	Values: `BR`, `SR`, `TR` — level of the linked requirement
`requirement_status`	Enum	—	SHALL	Values: `draft`, `approved`, `verified`, `deprecated`

`requirement_parent`	Reference	—	SHALL	Link to parent requirement (for hierarchy navigation)
`attempt_count`	Integer	SHOULD	SHALL	Number of implementation attempts (for FPSR calculation)
`created_at`	Timestamp	SHALL	SHALL	Auto-set on creation
`started_at`	Timestamp	SHALL	SHALL	Auto-set on transition to `active`
`completed_at`	Timestamp	SHALL	SHALL	Auto-set on transition to `done`
`session_id`	Reference	SHALL	SHALL	Which Session completed this Task
`supervisor_id`	Reference	—	SHALL	Who supervised this Task
`bypass_reason`	Text	SHOULD	SHALL	Justification if any Quality Gate was bypassed
`knowledge_refs`	List	—	SHOULD	Links to relevant Knowledge Base entries
`cross_deps`	List	—	SHALL	Cross-project dependency references
`cost`	Numeric	—	SHOULD	Token/API cost for this Task

1.2 Required Automations

Automation	Core/Foundation	Team+	Description
QG-0 field validation	SHALL	SHALL	Block transition to `active` unless `goal` and `acceptance_criteria` are non-empty
State transition timestamps	SHALL	SHALL	Auto-capture `started_at`, `completed_at` on state change
State transition rules	SHOULD	SHALL	Enforce valid transitions: `planning` -> `active` -> `done`; `blocked` can enter/exit from any state
Attempt counter increment	SHOULD	SHALL	Increment `attempt_count` each time Task returns from `done` to `active`
Story completion roll-up	—	SHALL	Auto-calculate Story status from child Task statuses
Increment progress tracking	—	SHALL	Dashboard showing Increment completion percentage
Bypass tracking	SHOULD	SHALL	Record which gate was bypassed, by whom, with what justification
Duplicate detection	—	SHOULD	Warn when a new Task goal closely matches an existing Task
Requirement orphan detection	—	SHOULD	Identify requirements without implementation Tasks and Tasks without requirements
Requirement change impact	—	SHOULD	When a BR/SR changes, list all affected downstream artifacts

1.3 Required Reports

Report	Core/Foundation	Team+	Description
Throughput	SHALL	SHALL	Tasks completed per Session, with trend over time
Lead Time distribution	SHALL	SHALL	Histogram of `completed_at - created_at`, segmented by complexity
FPSR	SHALL	SHALL	% of Tasks completed with `attempt_count = 1`
DER	SHALL	SHALL	% of Tasks where defects found after `done` status
Task status summary	SHALL	SHALL	Count of Tasks by status, filterable by Increment/Session
Cycle Time	—	SHALL	`completed_at - started_at` distribution
Cost per Task	—	SHALL	Total cost / Tasks done, segmented by complexity and work type
Cost Predictability	—	SHALL	Actual vs. planned cost per Increment
Bypass rate	—	SHALL	Gate bypasses / total gate evaluations, by gate type
Supervisor workload	—	SHOULD	Tasks per Supervisor per Session, with complexity weighting
Cross-dependency status	—	SHALL	Status of all cross-project dependencies

---

2. CI/CD Pipeline

Automated enforcement of QG-2 (Implementation Gate) and supporting infrastructure for QG-3 and QG-4.

2.1 QG-2: Implementation Gate (Automated)

All checks run on every commit or merge request. Pipeline must block merge if any check fails.

Check	Core/Foundation	Team+	Notes
Unit tests pass	SHALL	SHALL	100% pass rate required
Type checking passes	SHALL	SHALL	Language-appropriate: TypeScript strict, mypy, etc.
Linter passes	SHALL	SHALL	Zero warnings policy (not just errors)
Security scan (dependencies)	SHOULD	SHALL	Known CVE detection in dependencies
Security scan (SAST)	—	SHOULD	Static analysis for security anti-patterns
Build succeeds	SHALL	SHALL	Clean build from scratch, no cached state
Test coverage threshold	SHOULD	SHALL	Configurable minimum (recommendation: 70%+)
Dependency change detection	—	SHOULD	Flag new dependencies for review
Breaking change detection	—	SHOULD	API compatibility checks for shared interfaces

Pipeline configuration requirements:

Requirement	Core/Foundation	Team+
Pipeline runs automatically on push/MR	SHALL	SHALL
Pipeline blocks merge on failure	SHALL	SHALL
Pipeline results visible to Supervisor	SHALL	SHALL
No `allow_failure` — every job either works or is removed	SHALL	SHALL
Pipeline execution time < 10 minutes	SHOULD	SHALL
Pipeline provides clear failure messages	SHALL	SHALL
Pipeline is version-controlled (pipeline-as-code)	SHOULD	SHALL

2.2 QG-3: Verification Gate (Review)

Requirement	Core/Foundation	Team+	Notes
Branch protection on main/production branches	—	SHALL	No direct push
Merge request required	—	SHALL	All changes go through MR
Minimum 1 approval required	—	SHALL	Approver must be different from the Supervisor who created the MR
Review checklist integration	—	SHOULD	AI Review Checklist (Guide 02) embedded in MR template
Approval cannot be from the MR author	—	SHALL	Enforce independent verification
Stale approval invalidation	—	SHOULD	Re-approve after new commits

2.3 QG-4: Acceptance Gate (Deployment)

Requirement	Core/Foundation	Team+	Notes
Staging environment exists	—	SHALL	Mirrors production configuration
Staging deployment is automated	—	SHALL	Same pipeline as production, different target
Staging verification step	—	SHALL	Manual or automated acceptance test on staging
Production deployment requires explicit approval	—	SHALL	Human gate — not auto-deploy on merge
Rollback capability	—	SHALL	Ability to revert to previous version within minutes
Deployment tracking	—	SHALL	Record what was deployed, when, by whom
Smoke tests post-deployment	—	SHOULD	Automated verification that deployment succeeded
Feature flags	—	SHOULD	Ability to disable features without redeployment

---

3. Knowledge Base

The Knowledge Base is the persistent memory of the project. AI Agents have no memory between sessions — the KB is how knowledge survives.

3.1 Core Requirements

Requirement	Core/Foundation	Team+	Notes
Searchable (full-text)	SHALL	SHALL	Supervisor and AI Agent must be able to search by keyword
Categorized by entry type	SHALL	SHALL	Minimum types: `decision`, `pattern`, `gotcha`, `dead_end`, `observation`
Entry has title and body	SHALL	SHALL	Title for scanning, body for detail
Entry has creation timestamp	SHALL	SHALL	When was this knowledge created
Entry has project/scope tag	SHOULD	SHALL	Which project or area this knowledge applies to
Versioned entries	—	SHALL	Track changes to knowledge entries over time
Freshness tracking	—	SHALL	`last_reviewed` timestamp, updated when entry is confirmed still accurate
Entry status	—	SHALL	Values: `current`, `needs_review`, `deprecated`
Cross-references between entries	—	SHOULD	Link related knowledge entries
Bulk export	SHOULD	SHALL	Export all entries in a portable format (JSON, markdown)
Entry author tracking	—	SHALL	Who created or last updated this entry

3.2 Entry Types

Type	Purpose	Example
`decision`	Architectural or design choice with rationale	"We use UUIDs for all entity IDs because..."
`pattern`	Reusable approach for common tasks	"Error handling in API endpoints follows this structure..."
`gotcha`	Non-obvious behavior that causes problems	"CouchDB bulk_docs silently ignores conflicts unless..."
`dead_end`	Approach that was tried and abandoned, with reason	"Tried using WebSockets for SSE replacement — failed because..."
`observation`	Empirical finding worth remembering	"Build times increase 2x when running TypeScript strict mode on..."
`template`	Reusable requirement/AC pattern for common task types	"REST API endpoint AC: 1. Returns 200 for valid input. 2. Returns 401 without auth..."

3.3 AI Agent Integration

Requirement	Core/Foundation	Team+	Notes
AI Agent can search KB during session	SHALL	SHALL	Via tool/function call or context injection
AI Agent can create KB entries during session	SHALL	SHALL	Dead Ends and Gotchas discovered during work
Relevant KB entries injected into session context	SHOULD	SHALL	Based on task area, project, or explicit references
AI Agent can update existing KB entries	—	SHALL	Correct or extend existing knowledge
KB query results are structured	SHOULD	SHALL	Title, type, body, freshness — not raw text dumps
KB access is scoped	—	SHALL	Agent sees relevant project knowledge, not everything
KB entry creation triggers review queue	—	SHOULD	Human reviews AI-created knowledge entries

3.4 Freshness Management

Requirement	Core/Foundation	Team+	Notes
Entries older than N increments flagged for review	—	SHALL	Configurable threshold (recommendation: 3 Increments)
Freshness report available	—	SHALL	Distribution of entry ages, count by status
Deprecated entries excluded from AI context	—	SHALL	Stale context is worse than no context
Quality Sweep includes KB freshness audit	SHOULD	SHALL	Part of the ceremony checklist

---

4. Session Management

Sessions are SENAR's primary rhythm mechanism. The tooling must support the full session lifecycle.

4.1 Session Lifecycle

Requirement	Core/Foundation	Team+	Notes
Session start with timestamp	SHALL	SHALL	Records when the Session began
Session end with timestamp	SHALL	SHALL	Records when the Session ended
Session has unique identifier	SHALL	SHALL	For linking Tasks, metrics, handoffs
Session links to Supervisor	SHOULD	SHALL	Who ran this Session
Session links to Tasks worked	SHALL	SHALL	Which Tasks were active during this Session
Previous session handoff available at start	SHALL	SHALL	Context continuity

4.2 Checkpoint Capability

Requirement	Core/Foundation	Team+	Notes
Mid-session checkpoint	SHOULD	SHALL	Save context state without ending the Session
Checkpoint captures current task status	SHOULD	SHALL	What's in progress, what's done
Checkpoint reminder	—	SHOULD	Alert at configurable intervals (e.g., every 40-60 tool calls)
Checkpoint restores context on session crash	—	SHOULD	Recover from unexpected session termination

4.3 Handoff Documents

Requirement	Core/Foundation	Team+	Notes
Handoff document created at Session End	SHALL	SHALL	Summary of what was done and what's next
Handoff includes completed Tasks	SHALL	SHALL	List of Tasks done in this Session
Handoff includes in-progress Tasks	SHALL	SHALL	State of Tasks not finished
Handoff includes blockers/risks	SHOULD	SHALL	What might prevent the next Session from succeeding
Handoff includes knowledge captured	—	SHALL	KB entries created or updated during the Session
Handoff stored and searchable	SHOULD	SHALL	Historical handoffs available for reference
Handoff loaded automatically at next Session Start	SHOULD	SHALL	No manual searching for the latest handoff

4.4 Metrics Capture

Requirement	Core/Foundation	Team+	Notes
Tool call count per Session	SHOULD	SHALL	Leading indicator of session complexity and fatigue
Session duration	SHALL	SHALL	Derived from start/end timestamps
Tasks completed per Session	SHALL	SHALL	Throughput input
Session cost (tokens/API spend)	—	SHALL	For Cost per Task calculation
Error/retry count	—	SHOULD	How many times the AI needed correction

---

5. AI Agent Integration

Requirements for how AI Agents are configured, monitored, and managed within SENAR.

5.1 Context Injection

Requirement	Core/Foundation	Team+	Notes
Structured project instructions	SHALL	SHALL	Project-level context loaded at session start (e.g., CLAUDE.md, .cursorrules)
Task context injection	SHALL	SHALL	Goal, AC, and relevant constraints provided to agent
KB entry injection	SHOULD	SHALL	Relevant knowledge entries available during work
Codebase conventions	SHOULD	SHALL	Coding standards, architectural patterns, naming conventions
Scope boundaries	SHOULD	SHALL	Explicit "do not change" instructions
Session history/handoff injection	SHOULD	SHALL	Previous session context for continuity

5.2 Output Capture

Requirement	Core/Foundation	Team+	Notes
Code changes captured in version control	SHALL	SHALL	Standard git workflow
Session logs available for review	SHOULD	SHALL	What the AI did, what the Supervisor approved
AI reasoning visible	SHOULD	SHOULD	Why the AI made specific choices (for verification)
Output linked to Task	SHALL	SHALL	Which Task produced which code changes

5.3 Cost and Token Tracking

Requirement	Core/Foundation	Team+	Notes
Token usage per session	—	SHALL	Input + output tokens
API cost per session	—	SHALL	Dollar cost derived from token usage
Cost per task (derived)	—	SHALL	Session cost / tasks completed
Cost trend over time	—	SHALL	Is the process becoming more or less expensive?
Budget alerts	—	SHOULD	Warn when session/increment cost approaches budget
Cost by model/provider	—	SHOULD	If multiple models are used, track cost separately

5.4 Multi-Agent Orchestration (Team+)

Requirement	Core/Foundation	Team+	Notes
Workspace isolation	—	SHOULD	Agents work in isolated environments (worktrees, containers)
Agent-to-agent communication	—	SHOULD	Delegate subtasks to specialized agents
Parallel agent execution	—	SHOULD	Multiple agents working simultaneously under supervision
Agent output aggregation	—	SHOULD	Collect results from multiple agents into unified review
Agent capability profiles	—	SHOULD	Different agents for different task types (code, test, docs)

5.5 Agent Configuration Management

Requirements for managing Agent Profiles, Operational Scripts, and the Programmatic Interface (Standard Section 5).

Requirement	Core/Foundation	Team+	Notes
Script storage in version control	SHALL	SHALL	Scripts stored alongside project code
Script versioning (change history)	SHOULD	SHALL	Track who changed what, when, and why
Profile definition storage	SHOULD	SHALL	Agent Profiles stored as structured configuration
Profile switching support	—	SHOULD	Ability to switch agent context between profiles within a session
Permission scoping per profile	—	SHALL	Enforce read/write boundaries per Agent Profile
Script propagation mechanism	—	SHALL	Propagate script changes across multiple projects
Script rollback capability	SHOULD	SHALL	Revert any script change to previous version
Active script registry	—	SHALL	Inventory of active scripts with version identifiers
Script change audit trail	—	SHALL	Log of all script modifications with rationale
Tool inventory per profile	SHOULD	SHALL	Document which tools each profile can access
Hook execution engine	—	SHOULD	Automated actions triggered by events (post-session, pre-commit)
Script effectiveness metrics	—	SHOULD	Track FPSR and error rate per script version

---

6. Integration Requirements

How the components connect to form a functioning SENAR toolchain.

6.1 Core/Foundation Integrations

Integration	Description
Task Tracker <-> Session Management	Sessions reference Tasks; Tasks reference Sessions
Task Tracker <-> CI/CD	Pipeline results linked to Tasks and merge requests
Knowledge Base <-> AI Agent	Agent can search, read, and create KB entries during sessions
Session Management <-> AI Agent	Session start/end driven by or coordinated with agent lifecycle

6.2 Team+ Integrations

Integration	Description
Task Tracker <-> Knowledge Base	Tasks link to relevant KB entries; KB entries link to originating Tasks
CI/CD <-> Task Tracker	Pipeline failures auto-update Task status to `blocked`
Session Management <-> Metrics	Session metrics auto-populated from session data
Knowledge Base <-> CI/CD	Architectural constraint KB entries enforced in pipeline (aspirational)
Federation <-> Task Tracker	Cross-project dependency status synchronized
Federation <-> Knowledge Base	Cross-project knowledge accessible with proper scoping

---

7. Selection Guidance

When evaluating tooling options, prioritize:

1. **Automation over manual process** — if a ceremony step can be automated, it should be. Manual steps are failure points (see Guide 06, PF-3: Gate Bypass Normalization).

2. **Portability over features** — data must be exportable. Vendor lock-in (Guide 06, OF-3) is a recognized failure mode.
3. **AI-accessible over human-only** — every data store the Supervisor uses should also be queryable by the AI Agent. If the AI can't access it, it doesn't exist for SENAR purposes.
4. **Simple over comprehensive** — a task tracker with 5 well-implemented fields beats one with 50 fields nobody fills in. Start with Core/Foundation requirements and add Team+ requirements when needed.
5. **Composable over monolithic** — prefer tools that integrate via APIs over all-in-one platforms. SENAR components evolve at different rates; replacing one component should not require replacing all of them.

SENAR Reference: Code Standards Template

> *Organizations SHOULD maintain a Code Standards document loaded into every AI agent's context (Rule 15, L2). This template provides a starting point derived from adversarial audits of production AI-generated code.*

How to Use

1. Copy this template to your project's agent configuration (e.g., `.claude/references/code-standards.md`)
2. Adapt rules to your stack (Python, TypeScript, Go, etc.)
3. Add project-specific conventions
4. Ensure the document is loaded into every agent session automatically

Template Sections

A Code Standards document SHOULD cover:

Section	What to Define	Example Rules
Security	Input validation, access control, header trust, injection prevention	Never trust HTTP headers for security decisions without proxy validation
Architecture	File/function size limits, SRP, dependency injection	Max 400 lines/file, max 50 lines/function
Database	Query safety, migration patterns, bounded queries	Always use parameterized queries; always set LIMIT
API	Auth checks, schema validation, error handling	Every endpoint: verify auth AND resource access
Concurrency	Thread safety, resource lifecycle, data immutability	Shared mutable state requires locks
Domain-specific	LLM output validation, event processing, etc.	Validate LLM output before downstream use
Testing	Regression tests, mock boundaries, assertion quality	Every fix MUST include a regression test
Configuration	Startup validation, secure defaults	Fail fast if security config is missing

Key Principle

Every rule in a Code Standards document SHOULD be traceable to a real defect. Theoretical rules are ignored; evidence-based rules are followed.

See SENAR Guide Chapter 2 (AI Output Review Checklist) for complementary runtime checks.