

# Generative Specification: The Practitioner's Protocol

# Generative Specification: The Practitioner's Protocol

**Status:** Living Document

**Version:** 1.3 — April 2026

**Companion to:** *Generative Specification: A Pragmatic Programming Paradigm for the Stateless Reader*

---

## Preface

This document is the execution guide. The white paper establishes the structural argument — why externalized, derivable specifications are the necessary grammar for AI-assisted software systems, and why the failure mode of leaving intent implicit propagates at generation speed. That argument belongs in the paper.

This document answers the practitioner's question: *what do I actually do?*

Everything here is procedural. If a claim belongs in a paradigm argument rather than an instruction, it belongs in the paper. If it tells you what file to create, in what order, with what contents, and when to stop — it belongs here.

The document is organized around a single cognitive model: the **five memory types** that a generative specification system must provide for its stateless reader. Use the taxonomy to diagnose a project's artifact set. A project with no episodic record is working from amnesia. A project with no relationship memory will drift into incoherence at every boundary. The taxonomy tells you what's missing and what it costs.

**On practitioner evolution.** Expect heavy AI dialogue early — especially on your first few GS projects. You are learning the domain while writing the spec; the AI helps surface what you don't yet know to ask. This is correct practice, not a sign of inadequate specification. As your fluency builds, the dialogue contracts: later projects require only course corrections and directed expansions. The AI's role shifts from collaborator to executor. If you are still having the same kind of conversations after ten projects that you had on your first, the spec is incomplete — the dialogue is compensating for a gap that should be in the artifact set. The direction of expertise is: prompt less, specify more exactly.

**Paper reference:** This document is the companion methodology documentation explicitly referenced in §§6.5, 6.6, and 6.7 of the white paper (`GenerativeSpecification_WhitePaper.md`). The paper establishes the structural argument; this document provides the execution protocol.

---

## Part I: The Obligation Cascade

### 0. What GS Removes — and What It Adds

Every tier of GS operates on two axes simultaneously: it adds a restriction to the system and removes an obligation from the practitioner. These are the same move stated from two directions. Understanding both axes is what distinguishes a practitioner applying the discipline from one merely following procedures.

Tier	Restriction added	Obligation removed	Status
<b>T1 — Specification</b>	All valid implementations must be derivable from the spec	You do not write code	Demonstrated in production + DX study
<b>T2 — Harness</b>	No output is accepted without passing the behavioral harness	You do not read generated code	Demonstrated: AX adversarial series + DX study
<b>T3 — Infrastructure</b>	All infrastructure state must be in the spec	You do not touch infrastructure manually	Demonstrated: production regulated platform (COMPASS ETL)
<b>T4 — Monitoring</b>	Runtime drift is a specification violation	You do not diagnose bugs	Demonstrated: COMPASS/The Eye diagnostic agent
<b>T5 — Evolution</b>	All system evolution must stay within the telos	You do not maintain the system	Demonstrated: Loom colony, governed genome mutation running ( <a href="https://github.com/jghiringhelli/loom">github.com/jghiringhelli/loom</a> )
<b>T6 — Synthesis</b>	The entire architecture must derive from the problem statement	You do not design the system	Conclave architecture; research frontier
<b>T7 — Meta-telos</b>	A system observing the practitioner infers what is needed	You do not initiate the process	Research agenda; governance prerequisite

**The cascade is not a checklist to reach the end of.** T1 and T2 are the entry point for every practitioner. T3 is achievable without specialized tooling — it requires complete NFRs at T1. T4 requires T3 to be complete. **T3 and T4 failures almost always trace to T1 gaps: an NFR not stated, a data flow label missing, a contract left implicit.** The cascade is conceptually hierarchical; the correction loop is recursive. When something breaks at T3 or T4, the fix is in the spec, not in the infrastructure.

**On the harness.** The specification establishes intent. The harness certifies the derivation was faithful. A specification without a T2 harness is an assertion, not a guarantee — “spec is the program” holds only when the behavioral contracts are continuously verified against the running system. T2 is not optional scaffolding. It is constitutive of the GS guarantee.

**Starting at T1 is correct.** Most practitioners run T1 and T2 indefinitely and reach excellent results. T3 becomes relevant when infrastructure state is complex enough to drift. T4 becomes relevant when the system is in production long enough to accumulate observable behavior. Do not force the cascade depth — let the project’s failure modes tell you which tier to activate next.

## Part I: The Cognitive Framework

### 1. The Five Memory Types

An AI assistant has no persistent memory across sessions. Its context window — bounded, populated at session start by what the practitioner loads, reset at session end — is its entirety of cognitive access. This is not a limitation to work around. It is the constraint the entire methodology is designed to address structurally, rather than by hoping the next session starts with a sufficiently rich context.

The methodology distributes memory across five artifact classes, each serving a distinct cognitive function.<sup>[1](#)</sup>

Memory Type	Cognitive Function	Primary Artifacts
Semantic	What the system <i>is</i> — its identity, contracts, and constraints	CLAUDE.md, tech spec, domain models, glossary
Procedural	How things are done — execution rules, pipelines, bound prompts	DEVELOPMENT_PROMPTS.md, roadmap, CI/CD spec, commit hooks
Episodic	What <i>happened</i> — decisions made, sessions completed, history recovered	ADRs, Status.md, session summaries, git commit log
Relationship	How things <i>connect</i> — component topology, flows, protocols	C4 diagrams, sequence diagrams, state machines, use cases
Working	What is <i>active now</i> — current task, loaded context, session scope	Session prompt, loaded artifacts, clarification state

Every artifact in a well-formed generative specification belongs to exactly one of these types. When an artifact is ambiguous about which type it serves, it is trying to do too much and will do none well.

**The taxonomy as diagnostic tool.** Run this check on any project before beginning methodology work:

- Semantic artifacts absent → the AI has no grammar; output will be locally correct and globally incoherent.
- Procedural artifacts absent → each session starts from scratch; nothing is reproducible.
- Episodic artifacts absent → decisions are repeated or overwritten; the AI will “improve” intentional choices.
- Relationship artifacts absent → inter-component contracts are implicit; integration points will drift.
- Working artifacts absent (or not loaded) → the current session inherits no context from the previous one; the practitioner re-narrates everything.

Each missing type compounds. A project missing all five is not using generative specification — it is using interactive prompting with no structural discipline.

Tulving, E. (1972). Episodic and semantic memory. In E. Tulving & W. Donaldson (Eds.), *Organization of Memory* (pp. 381–403). Academic Press. Tulving, E. (1985). Memory and consciousness. *Canadian Psychology*, 26(1), 1–12. Squire, L.R. (1987). *Memory and Brain*. Oxford University Press.

## Part II: Semantic Memory — What the System Is

### 2. The Architectural Constitution

The architectural constitution (CLAUDE.md) is the primary semantic artifact. It is the grammar the AI reads before any session prompt, and it persists across every session. Everything the AI is not supposed to do, and everything it must do regardless of what a session prompt says, lives here.

**What it must contain:**

- Target architecture: layer names, allowed dependency directions, forbidden patterns (no direct DB calls from UI, no bare exception throws, no module-level instances)
- Quality gates: coverage threshold (as a number), lint rules, complexity ceiling
- Naming conventions: enough specificity to distinguish layer ownership from name alone (findUserByEmail vs getUserProfile)
- Error handling standard: exception hierarchy, required context fields, where HTTP codes may appear
- Technology decisions: locked versions, approved libraries, explicitly forbidden libraries

- Guard clauses: the single most common AI deviation — writing nested conditionals rather than early-return guards — must be stated explicitly as forbidden, with the correct pattern shown
- Commit format: the conventional commit prefix set, scope conventions, what triggers a version bump

### What it must not contain:

- Tutorial content (the AI already knows how tests work)
- Motivation for obvious rules
- Scope that has drifted from the current project (the AI reads everything; stale rules are as binding as current ones)

**Constitution hygiene — the compression protocol.** A constitution that grows without discipline defeats itself. The AI reads it in full on every turn; at depth, attention distributes less precisely and relevant rules are diluted by bulk. The threshold is approximately 250–300 lines. When the document approaches this ceiling:

1. Run `setup_project` with tier: core to compress to essentials — ForgeCraft preserves custom sections and the Corrections Log.
2. Alternatively, audit manually: every section that repeats general best practices (rather than project-specific constraints) is a candidate for removal. SOLID principles belong in `CLAUDE.md` only as project-specific rules, not as tutorial content. (GS is a pragmatic-tier discipline; SOLID and TDD are semantic-tier disciplines that operate at a different altitude — they govern how code is structured, not whether the spec is the primary artifact. A constitution that tutorializes semantic-tier disciplines is doing it wrong: the AI already knows them. Name them in the Techniques subsection as activation keys; do not explain them.)
3. If scope has drifted (new tag category added, framework changed), run `refresh_project` first; it detects tag drift and regenerates cleanly before compressing.

**The self-healing hook.** An architectural constitution without enforcement is advisory. The pre-commit hook is the enforcement mechanism: it runs the full lint + type-check + unit test suite before every commit, blocking on failure. The AI maintains the hook as part of the constitution; it cannot remove or weaken the gate without a documented ADR.

Implementation for a TypeScript project:

```
# .husky/pre-commit
#!/usr/bin/env sh
. "$(dirname -- "$0")/_/husky.sh"
npm run lint && npm run typecheck && npm run test:unit
```

For Python:

```
#!/usr/bin/env sh
ruff check . && mypy . && pytest tests/unit/
```

The hook specification — what commands run, in what order, what failure means — belongs in the constitution, not in the hook file alone. The hook is the executor. The constitution is the authority.

**The Corrections Log.** The architectural constitution has one required section that every practitioner misses on first pass: the Corrections Log. Every time the practitioner corrects the AI's output — a deviated pattern, a violated convention, a misapplied rule — the correction is recorded as a one-line entry at the bottom of `CLAUDE.md`. This is not optional documentation. It is a feedback loop written into the grammar itself.

Minimum section format (place at the bottom of `CLAUDE.md`):

```
## Corrections Log
When I correct your output, record the correction pattern here so you don't repeat it.
```

### ### Learned Corrections

- [AI assistant appends corrections here with date and description]

Trigger rule: any session where the practitioner says “don’t do that” about a pattern the AI produced is a Corrections Log event. The AI appends the entry immediately. Format: [YYYY-MM-DD] – [pattern corrected]. Example: [2026-03-12] – Do not wrap early-return guards in positive outer conditions; handle invalid cases at the top of every function.

ForgeCraft’s setup\_project and refresh\_project preserve the Corrections Log section when regenerating or compressing the constitution.

**The Techniques Subsection.** The architectural constitution is not only an architectural grammar — it is a technique registry. A named technique in the specification is available to the AI at the full depth of its training on that technique. RAPTOR indexing traveled from Conclave to BRAD to SafetyCorePro not because any session carried it forward in context, but because each project’s constitution named it explicitly.

Maintain a **Techniques** section in CLAUDE.md listing every named method, algorithm, framework, or domain-specific pattern the project uses or should use when relevant. Examples: RAPTOR indexing (hierarchical codebase and document summarization), BM25 + vector hybrid retrieval with RRF fusion, PCA-based geometric validation for generative assets, deontic modal logic for obligation/permission domain modeling. The AI does not need these explained in the constitution. It needs them named. The name is the activation key.

**The Known Pitfalls Subsection.** Every technology stack accumulates a set of traps that are not obvious from documentation alone — library type definitions that differ from their runtime behavior, tool flags that silently no-op, minimum constraints that cause startup failures at runtime but not at compile time. These recur across sessions because the AI cannot remember that the trap was already hit. The Known Pitfalls section makes them pre-emptive: the AI sees the documented trap before writing the code that would fall into it.

Maintain a **Known Pitfalls** section in CLAUDE.md. Each entry has three parts: what goes wrong and why, the wrong pattern, and the correct pattern. Entries are added when a trap is hit in any session — they belong here, not in the Corrections Log, when the issue is a general library/tooling characteristic rather than a project-specific deviation.

Example entry structure:

### ## Known Pitfalls

#### ### `prisma db push` vs `prisma migrate deploy` in test environments

``migrate deploy`` requires a pre-existing ``prisma/migrations/`` folder. When none exists, it silently succeeds (no-op), leaving the database empty. All integration tests then fail with table-not-found errors – a ghost failure cascade that obscures the real problem.

✗ Wrong (silently no-ops with no migration files):

``npx prisma migrate deploy``

✓ Correct for test environments (syncs schema.prisma directly to the DB):

``npx prisma db push --accept-data-loss``

The Corrections Log captures AI behavioral deviations. The Known Pitfalls section captures technology traps. They are distinct artifacts with distinct triggers.

## 3. The Technical Specification and Domain Models

The architectural constitution governs *how* the system is built. The technical specification governs *what* it is and *what it must do*.

## Minimum contents of a technical specification:

- Functional scope: what the system does, at the level of user-visible behavior
- Data models: entity definitions, field types, constraints, relationships — stated with enough precision that two different engineers writing separate modules would produce compatible schemas
- API contracts: endpoint signatures, expected inputs and outputs, error shapes, authentication requirements
- Non-functional requirements: latency targets (p99), throughput ceiling, availability SLA, data retention policy, security compliance requirements

**Naming as a contract.** In a polyglot system, naming is the cross-language contract. A function named identically in a TypeScript API surface and its Python backend will not produce incoherent output when the AI touches both sides. A function named `processItem` in TypeScript and `handle_document` in Python for the same concept will. The technical specification names every domain concept once, in language-neutral terms. Both sides derive from the same vocabulary.

The BRAD case study demonstrates this at the domain level: naming prosody analysis, argumentation theory, fallacy classification, and deontic modal logic in the specification activates domain knowledge that a generic “analyze arguments” instruction does not. The specification is not just an architectural grammar — it is a technique registry whose scope is the full depth of the model’s training.

**Platform-independent migration specification.** When migrating an existing system to a new platform, the specification step extracts behavioral contracts from the source system in platform-neutral terms before any new code is written. The specification describes what each system does, not how it does it in the source platform. A Unity/C# behavior specification that does not mention `MonoBehaviour`, `GameObject`, or `SerializeField` is portable. One that does is tied to the old executor.

The broken implementation becomes irrelevant. A broken implementation is a complete specification with a bad executor. The methodology replaces the executor.

---

## Part III: Procedural Memory — How Things Are Done

### 4. The Initialization Cascade

The initialization cascade runs exactly once per project, in this order. Each step is derived from the one above it. No step proceeds before the one above it is complete.

**Step 1: Functional specification.** Write a prose description of what the system does, for whom, and what constitutes success. This is not a user story. It is a precise statement of scope: what the system includes and — equally important — what it explicitly excludes. Ambiguous scope here propagates into every derived artifact. Spend time here. The AI cannot fix an underspecified intent.

**Step 2: Architecture and C4 diagrams.** From the functional specification, derive the system architecture: the component topology (C4 context and container levels), the inter-service boundary contracts, and the data flow. This does not require implementation to exist. It requires a decision. The C4 context diagram names what the system is and what it communicates with. The container diagram names the deployable units and their boundaries. At this step, sequence diagrams for the primary flows are sketched — they become precise in step 4.

**Step 3: Architectural constitution.** From the architecture, derive the CLAUDE.md: the layer rules, naming conventions, forbidden patterns, quality gates, and technology decisions that govern every session that follows. This is the grammar the AI reads before every prompt. It must describe the architecture decided in step 2 — not a generic best-practice document.

Generate with ForgeCraft (setup\_project), then review and customize. ForgeCraft auto-detects tags from the project structure and produces a constitution covering the relevant domain standards. The practitioner’s customization addresses what ForgeCraft cannot detect: project-specific naming conventions, technology choices with specific version constraints, and domain-specific invariants.

**The sentinel workflow.** ForgeCraft exposes a single MCP tool — the sentinel — that reads three artifacts (project configuration, constitution, hooks), derives the correct next action, and returns one CLI command. Its token cost is approximately 200 tokens of context per turn, compared to ~1,500 tokens per tool in a conventional multi-tool surface. The recommended workflow: add the sentinel at project initialization, run setup\_project to generate the constitution and hooks, then optionally remove the sentinel from the active MCP server list to reclaim context budget for the implementation phase. The sentinel can be re-enabled at any time as a lightweight drift detector — it reads the current artifact state and diagnoses what is missing or misconfigured. The “add → setup → remove” cycle is itself a GS hygiene rule: load only what the current session needs.

**The sentinel navigational tree — completeness requirement.** The sentinel reads CLAUDE.md to derive the project’s governing context. For the sentinel to function correctly — and for any AI session to inherit a complete grammar — the architectural constitution must provide coverage across five categories. A constitution missing any category will produce sessions that drift in the corresponding dimension:

Category	What it provides	Drift when absent
Architectural identity	What the system is, its boundaries, its primary purpose, its tech stack	AI treats each session as a blank-slate design problem
Standards	Named patterns, disciplines, and protocols the project follows (SOLID, conventional commits, RAPTOR, BM25+vector)	Sessions vary technique application; named disciplines are not activated
Constraints and prohibitions	Explicit forbidden patterns (goto-equivalents, direct DB calls from routes, bare exception throws, hardcoded values)	AI optimizes locally, violates boundary rules without prompting
Tool sequencing	Ordered tooling instructions — what to install, what to run, in what order, what each failure means	Sessions reinvent setup; tool failure cascades misdiagnosed
Routing	Which artifact to consult for which class of decision — spec for contracts, ADR for architecture history, use cases for behavior	AI ignores available context; makes decisions already recorded elsewhere

Audit a CLAUDE.md against this five-category grid before the first implementation session. Missing categories are not style choices — they are specification gaps with predictable failure modes.

**Step 4: ADR initialization.** For every non-obvious decision made in steps 1–3 — technology selection, architectural pattern, exclusion of scope — write an ADR before any implementation begins. An ADR at initialization is the rationale record for decisions whose alternatives were considered and rejected. Without it, the AI will “optimize” these decisions in a future session.

Minimum ADR format (canonical template in §9):

```
# ADR-NNN: Title
**Status:** Accepted  **Date:** YYYY-MM-DD
## Decision – one sentence.
## Context – problem addressed; alternatives considered and rejected.
## Consequences – what this enables; what it forecloses.
```

**Step 5: Use cases and session-scoped prompts.** From the functional specification and architecture, derive the use cases (actor, precondition, trigger, postcondition, error cases) and the bound roadmap. Each roadmap item receives a session-scoped prompt before any implementation session begins (see §5 for bound prompt format).

**Derivability gate.** The initialization cascade is complete when a stateless agent given the five artifact sets — functional spec, architecture diagrams, architectural constitution, ADRs, use cases+prompts — can derive any valid implementation state without further human direction. Apply this test before proceeding to implementation: if you would need to narrate something that isn't in the artifacts, the cascade is not complete.

**Scoping the initialization cascade — the MVP entry path.** The initialization cascade does not require the full system to be specified before the first session begins. It requires the *current session's scope* to be completely specified before generation starts. A complete specification for the minimum viable system is still a complete specification. “Complete” means: every element within the declared scope is specified; every element outside it is explicitly excluded or deferred in an ADR.

The sequence for incremental entry: (1) Define the smallest scope that produces observable value. (2) Run the full initialization cascade for that scope. (3) Implement. (4) When the scope expands — new feature area, new integration, new data flow — write the expansion ADR first, then extend the specification, then open the implementation session. Each scope expansion is a mini-cascade: spec update → ADR → implementation. Never expand scope within an implementation session without closing the previous scope first.

This pattern is not a compromise on discipline — it is the discipline applied correctly. The cascade gates ensure that every tier is complete for whatever scope is in flight. A practitioner who “will write the spec later” has not applied incremental entry; they have applied deferred specification, which is the failure mode the discipline exists to prevent.

## 5. The Bound Roadmap and Session-Scoped Prompts

A roadmap item without a pre-generated prompt is a task title. It requires the practitioner to reconstruct context at execution time. A roadmap item *with* a bound prompt is an independent execution unit: it already contains the specification references, the acceptance criteria, and the verification steps. The agent receives the prompt alongside the live spec artifacts and executes without further elaboration.

### Bound prompt format:

## [Prompt ID] – [Feature or Task Name]

```

**Specification references:** [List of artifacts the agent should load before beginning]
**Precondition:** [What must be true before this task begins]
**Scope:** [Explicit list of what to build and what NOT to touch]
**Acceptance criteria:**
- [ ] [Specific, verifiable criterion]
- [ ] [Full test suite passes]
- [ ] [Feature exercised at HTTP or CLI boundary]
**Architecture constraints:** [Any layer rules or patterns to enforce for this specific task]
**Commit message:** feat([scope]): [description]

```

The specificity is non-negotiable. “Build the connection system” is not a bound prompt. “Implement the createConnection service method against the ConnectionRepository interface, write unit tests for all three error paths, verify via Playwright test against the /connections endpoint, then run the full suite before committing” is a bound prompt.

**The waiting state protocol.** A project whose current roadmap item has a bound prompt and whose previous commit passes the full test suite is in a valid waiting state. The next session can begin immediately from the bound prompt without any narration. A project in a waiting state requires no execution from the practitioner — it



is simply not the active project in the current session. The portfolio is a collection of projects at various stages of readiness. A waiting state is not a blocked state.

## 6. The Incremental Cascade

Every change to a running system consists of two decisions: *what* to change, and *which artifacts* that change affects. The incremental cascade manages the second decision.

### Full procedure:

1. **Name the delta.** Observe something: a discrepancy in the running system, a new requirement, a design insight, a bug. Write it down in precise terms. Vague deltas produce vague cascades.
2. **Impact assessment (CIA).** Before propagating anything, answer: Which artifacts reference the changed element? Which roadmap items currently in progress share a dependency with it? Does the delta change a shared interface contract or schema? This is the Change Impact Assessment step. Skipping it risks propagating a change into implementation before discovering it breaks a parallel item already in progress.
3. **Determine minimum cascade depth.** Not every increment requires all five initialization steps.
  - A bug the existing specification correctly describes: implementation and closure only.
  - A new behavior within the existing architecture: update the use case, possibly the tech spec; skip C4 if no new components.
  - A new component or service boundary: update C4, the architectural constitution section covering the new component, the relevant ADR, and use cases before touching implementation.
  - A change to a shared interface contract: update the contract specification, the dependent use cases, and all affected roadmap prompts before any implementation.
4. **Propagate upward first, then downward.** The ordering constraint is absolute: when a layer needs updating, all layers above it are made consistent with the change *before* any layer below it receives it. Code that was correct relative to the old specification and is now wrong relative to the new one is a derivation gap, not a bug. The correct response is re-derivation, not patching.
5. **Close the cascade.** After implementation commits, run the documentation cascade (§7) to restore all artifacts to consistency.

### What “above” and “below” mean in practice:

```

Functional specification      ← top
↓
Architecture / C4 diagrams
↓
Architectural constitution (CLAUDE.md)
↓
ADRs
↓
Use cases / sequence diagrams
↓
Implementation + tests      ← bottom

```

Observing something in the running system (bottom) triggers the cascade upward to the minimum affected layer, then back down to implementation. Never skip a layer in the downward pass.

## 7. Loop Types and Gate Conditions

The methodology operates at four distinct loop granularities. Each has a defined gate condition. A loop that closes before its gate condition is met has not closed — it has paused.

**The initialization loop** runs once per project. Gate: the derivability criterion — a stateless agent given the complete artifact set can derive any valid implementation state without further narration.

**The incremental short loop** runs once per roadmap item or unscripted delta. Gate conditions, all of which must hold before the loop closes: - Full test suite passes, including coverage threshold - Feature is exercised at the HTTP or CLI boundary — not only unit-tested internally - Documentation cascade is complete - Status.md is updated with what was done, what is in progress, and what is next

External triggers — dependency CVEs, breaking upstream changes, API deprecations — are procedurally identical to any other spec delta: CIA first, then cascade at the depth the change warrants.

**The pre-release loop** runs before each environment promotion and is the methodology's hardening boundary. It requires deployment to at least one real environment — not a passing local suite. Gate: the release candidate criteria stated in the test architecture document — not a judgment call made at promotion time.

The hardening suite is ordered by when each activity requires a running environment:

**Pre-deployment (before staging deploy, on the candidate build):** - Full Stryker mutation run across the entire codebase — surviving mutants at this stage are test gaps that must be closed, not risks to absorb. MSI  $\geq 65\%$  overall,  $\geq 70\%$  on changed code. This is non-negotiable: shipping code with known test gaps is the same error as shipping with known lint failures, just harder to detect. - Full E2E suite on a local or ephemeral environment - `npm audit --audit-level=critical` (or equivalent) exits 0

**In-environment (after staging deploy):** - Smoke tests across all surfaces: every API entry point, every UI critical path, every database migration, every external dependency integration. Smoke failure at this stage indicates a deployment or configuration error — stop and diagnose before load testing. - Load tests: minimum parameters to specify are the target concurrent user population, the target throughput (requests per second), and the p99 latency ceiling. A “load test” without stated acceptance criteria is a manual observation. - Stress tests to failure: push beyond the load target until the system degrades or fails, document the failure point and recovery procedure. The failure mode is the deliverable — a system whose failure mode is undocumented has not been stress-tested. - DAST (dynamic security analysis) and penetration testing against the running environment. An in-repo OWASP ZAP or equivalent scan is the minimum; for systems handling authentication, payments, or PII, a human penetration test is not optional.

**Progressive rollout (if applicable):** Must name before the rollout begins: the canary population size, the error rate rollback threshold (absolute, not relative — “0.5% error rate”, not “2× baseline”), and the observation window. A rollout without these stated parameters is a full deployment with manual monitoring. That is not a rollout strategy.

**The hotfix loop** inverts the standard documentation order. A minimal targeted fix ships first. An ADR, the cascade artifact updates, and the rollback specification follow immediately after stabilization — not in the next scheduled session. The inversion is time-bounded: the documentation debt from a hotfix must be cleared before the next incremental loop begins.

**Knowing which loop you're in.** The four loops are parallel tracks at different altitudes sharing the same artifact set. Misidentifying the loop produces two failure modes: under-process (treating an architecture-level change as an incremental short loop, committing without cascade closure) and over-process (treating every bug fix as an initialization event, derailing a session into ceremony that a targeted fix did not require). The CIA step in the incremental cascade is the loop classifier: if the minimum cascade depth reaches C4 or the architectural constitution, you are closer to initialization territory than incremental territory.

**The code-generation verify loop.** Within the incremental short loop, each prompt response that produces implementation code requires a micro-loop before the practitioner accepts it. This is not documentation ceremony — it is the feedback mechanism that catches interface drift, incomplete fixes, and ghost failures before they compound across sessions.

The micro-loop procedure:

1. **Compile.** Run `tsc --noEmit` (TypeScript), `mypy` (Python), or the equivalent for the stack. Zero errors required.
2. **Test.** Run the full test suite. Zero failures required. Ghost failures — failures caused by infrastructure misconfiguration rather than code bugs — must be distinguished from real failures before building a fix prompt. A ghost failure cascade (e.g., all integration tests failing with “table does not exist” because the DB schema was never applied) fills the fix prompt with noise and produces a misleading diagnosis.
3. **Build the fix prompt with current file contents.** When errors remain, the fix prompt must include the current on-disk state of every file that the errors reference — not just the error messages. Providing error messages without the current source is the root cause of **interface drift**: the AI fixes one side of a call boundary (e.g., updates a service method signature) without seeing the other side (the route that calls the method), producing a consistent fix locally but inverting the same error on the next pass. Showing both sides simultaneously eliminates the oscillation.
4. **Include failing test files.** When tests fail, include the full content of the failing test files in the fix prompt. The AI cannot diagnose a `beforeAll` setup bug (e.g., a `$executeRawUnsafe` call with multi-statement SQL that pg rejects with error 42601) from test output alone. The test file contents make the diagnosis direct.
5. **Repeat until clean.** A verify loop that converges in 1–2 passes indicates adequate specification clarity. A loop that runs 5+ passes without convergence indicates either a ghost failure (check the infrastructure, not the code) or a structural mismatch between the fix prompt and what the model needs to see.

The verify loop is not a manual procedure — it is automated in any serious GS execution runner. The practitioner's role is to ensure the runner is built correctly: correct schema sync command (`db push`, not `migrate deploy`), sufficient fix prompt context (file contents, not just error messages), and bounded pass count with diagnostic output when the loop does not converge.

## 8. The Commit Discipline

A commit is a verified state of the system. Not a save. Not a checkpoint. A verified state.

**What a valid commit requires:** - Full test suite passes (enforced by pre-commit hook) - No new anti-patterns: no hardcoded values, no bare exception throws, no `console.log` left in production paths, no TODO/FIXME stubs returning hardcoded values - One logical change only — the delta is bounded and coherent - Conventional commit message: `feat|fix|refactor|docs|test|chore(scope): description`

**Commit message precision.** The commit message is the sentence describing this state in the typed corpus. `fix bug` is not a sentence. `fix(auth): reject expired tokens at middleware boundary before service layer invocation` is a sentence. The AI uses the commit history as context in future sessions. A history of no-op messages is not a corpus. A history of typed, scoped conventional messages is a queryable record of how the grammar evolved.

**What constitutes one logical change:** - A new feature and its tests: one commit - A refactor of an existing module that does not change behavior: one commit - A spec update (CLAUDE.md change + the code change it governs): one commit - A bug fix: one commit, with the failing test that caught it included in the same commit

**Anti-patterns to enforce via hook:**

```
# Detect hardcoded values (example – tune for your stack)
grep -rn "localhost|password|secret|api_key|http://" src/ --include="*.ts" | grep -v ".test."
```

The architectural constitution specifies what the hook checks. The hook executes the check. The practitioner maintains both artifacts in sync — a rule added to the constitution without a corresponding hook check is advisory, not enforced.

---

## Part IV: Episodic Memory — What Happened

### 9. Architecture Decision Records

Every non-obvious architectural decision produces an ADR before implementation begins. The test for “non-obvious” is not complexity — it is replaceability. If another engineer (or a future session of the current one) would reconsider the decision without knowing it was already made, it is non-obvious and requires an ADR.

**What triggers an ADR:** - Technology selection (why this library vs. the standard alternative) - Departure from the standard layer architecture - An exclusion of scope that another engineer would naturally include - A performance decision that sacrifices readability - A security tradeoff whose rationale must be preserved - A decision that is intentionally temporary — the ADR names when it should be revisited

**What does not trigger an ADR:** - Decisions where the alternative was not seriously considered - Implementation details that do not cross architectural layers - Style preferences already codified in the constitution

**ADR template (minimum viable):**

```
# ADR-NNN: Title

**Status:** Accepted
**Date:** YYYY-MM-DD

## Decision
[One clear sentence.]

## Context
[What problem this addresses. What alternatives were considered and rejected, and why.]

## Consequences
[What this enables. What it forecloses. What must change if this decision is reversed.]
```

Store in adr/ or docs/adr/. Number sequentially. Once accepted, an ADR is never edited — it is superseded by a new one (status: Superseded by ADR-NNN). The history of decisions is the episodic record of the architecture's evolution.

---

### 10. Status.md

Status.md is the episodic artifact closest to working memory. It is the artifact that answers “where was I?” It is updated at the close of every session, without exception.

**What it must contain:**

```
# Project Name – Status

**Last updated:** YYYY-MM-DD
**Current version / branch:**

## Completed (this session)
```

- [What was done, with commit hashes where relevant]

## ## In Progress

- [What is actively being built – any partial state, what the next step is]

## ## Next

- [The immediate next action – specific enough that an agent could begin from this line alone]

## ## Decisions made (this session)

- [Any choice made during the session that is not yet in an ADR – these are ADR candidates]

## ## Blockers / Dependencies

- [What is waiting on an external input or a parallel workstream]

The “Next” section is the handoff. If it is vague (“continue working on the feature”), the next session must reconstruct the intent. If it is specific (“implement `updateConnectionStatus` in `src/connections/service.ts`, write tests for the three state transition paths, verify against the `/connections/:id/status` endpoint”), the next session can begin from working memory position zero.

---

## 11. Session Summaries for Long Gaps

Status.md handles normal session-to-session continuity. When a project is in a waiting state for a week or more, the episodic record must carry more. A session summary adds:

- What the full system state is (not just what changed)
- What architectural constraints are active and why
- Where in the roadmap the project sits
- Any context that was live at session close but might not be obvious from Status.md alone

A session summary is distinct from Status.md in that it is not overwritten — it accumulates. Keep one per major phase. The AI reads it at session start for a project returning from long waiting. Without it, the session narration cost rises with the length of the gap.

---

## Part V: Relationship Memory — How Things Connect

### 12. Use Cases as Production Rules

A use case is not a requirements document. It is a multi-purpose production rule from which three things derive simultaneously without redundancy:

**Derivation 1 — Implementation contract.** A use case that names actor, precondition, trigger, and postcondition is the specification the service layer is written against. It answers: what state must the system be in before? what triggers the action? what state is the system in after? what constitutes an invalid call?

**Derivation 2 — Acceptance test.** The use case and the test scenario are the same artifact expressed in different dialects. A Playwright E2E test for a checkout flow is the checkout use case transcribed into executable form. When the use case is precise, the test writes itself. When the test is hard to write, the use case is underspecified. Test difficulty is the diagnostic for specification quality.

**Derivation 3 — User documentation.** A use case narrated to a non-technical reader — with actor, goal, sequence, expected outcome, and error cases — is a user manual section. The content is identical. The framing differs. A complete specification does not need a separate documentation writing pass; it needs a rendering pass.

**Use case format:**

```

## UC-NNN: [Action name]

**Actor:** [Who initiates]
**Precondition:** [What must be true before]
**Trigger:** [What event initiates the flow]
**Main flow:**
1. [Step by step – precise enough to be executable]
**Postcondition:** [What is true after successful completion]
**Error cases:**
- [Condition] → [System response]
**Out of scope:** [What this use case explicitly does not cover]

```

---

## 13. Diagram Types as Grammar Layers

Diagrams in a generative specification are not illustrations. They are constraints the AI reads before generating implementations.

**C4 — static structure.** Level 1 (context): what the system is and what it communicates with. Level 2 (containers): deployable units and their boundaries. Level 3 (components): internal structure of a container. Generate L1 and L2 at initialization. Generate L3 for complex containers. Keep them current through the ADR update cycle.

**Sequence diagrams — temporal and protocol contracts.** A sequence diagram specifying that authorization precedes data fetch — and not the reverse — is a stricter constraint than prose. The AI has two valid sentences in that portion of the grammar: the one matching the diagram, and deviations from it. Write sequence diagrams for every primary flow. For asynchronous flows, name the message queue and every consumer. For multi-hop flows, show every intermediate state.

**State machine diagrams — modal grammars.** Enumerate every valid system state and every valid transition. This is the source material for state transition tests and the documentation for any surface with modal behavior. A component that can be in an indeterminate state that no diagram names will accumulate bugs in that state until the diagram is drawn.

**User flow diagrams — behavioral grammar at the human layer.** The expected path through the system from the user's perspective. Simultaneously: the script for every E2E test in that flow, and the user journey narrative for the manual. Write these before the UI is built. A UI built without a user flow specification is a prototype. A UI built against one is an implementation.

---

## 14. Living Documentation

Documentation maintained separately from the code it describes drifts, structurally and inevitably. The methodology resolves this by treating documentation as derivation, not product:

- **OpenAPI from schema definitions.** TypeScript decorators (NestJS), Zod schemas with zodToJsonSchema, or FastAPI's automatic OpenAPI output — the schema definition is the source; the documentation is generated from it, not authored alongside it.
- **TypeDoc/JSDoc.** Inline documentation is the source. Published API reference is the output. No separate documentation writing pass. No documentation that can be wrong in a way that the code is right.
- **Storybook as component specification.** A Storybook story is simultaneously the component's specification, its usage example, and its visual test. Authored once; serves three consumers.

- **README sections from centralized specs.** README sections that pull directly from spec files rather than paraphrasing them cannot drift from the authoritative source.

**The polyglot case.** In a system spanning multiple languages, runtimes, and paradigms, the living documentation system is not optional enrichment. It is the only artifact that can hold inter-runtime contracts coherently. When the Python indexer's query interface changes, the TypeScript client update, the integration test rerun, and the documentation update for both are traceable from a single schema change — if the documentation is derived from the schema. If it isn't, the drift is guaranteed. Write the cross-language contracts in language-neutral terms, in a specification both sides derive from.

---

## Part VI: Working Memory — What's Active Now

### 15. The Session Loop

Every session begins and ends at the same steady state: code, tests, and documentation mutually consistent; the specification accurately describing what exists; the commit history recording how it changed; Status.md capturing intent for the session that follows. A session that ends with passing tests but a stale specification has not completed the loop.

**Phase 1 — Intake and clarification.** The practitioner voices intent. Before implementation begins, the agent checks for exactly two conditions: ambiguity (the request admits two or more valid interpretations that would produce different implementations) or unverifiable assumptions (the request presupposes facts about scope, schema, or behavior not verifiable from the current codebase). If either condition is present, one exchange resolves it — all clarifying questions batched into a single prompt, answered once. If neither condition is present, implementation proceeds immediately. The constraint on asking is as important as the obligation to ask.

**Phase 2 — Specification gate.** Before any code is written: does this change *fit* the existing specification, or does it *change* it? A feature the specification anticipates can be implemented directly. A feature the specification does not yet cover requires the specification to be updated first: the ADR, the schema change, the new section of the constitution. These precede implementation. Code written against the old specification is wrong by the grammar it was supposed to serve, regardless of whether the tests pass.

**Phase 3 — Implementation and verification.** Execute against the specification. Tests are written alongside the code they cover, not deferred. Before any commit, three conditions hold: 1. Full test suite passes 2. Feature exercised at HTTP or CLI boundary (not only unit-tested internally) 3. No new anti-patterns — no hardcoded values, no bare exception throws, no diagnostic logging in production paths

**Phase 4 — Documentation cascade.** After a passing commit, restore all artifacts to consistency, in this order: 1. Spec files (docs/specs/) — if a public contract changed (endpoint, CLI command, schema) 2. ADR — if a non-obvious architectural decision was made 3. Architectural constitution — if a new pattern was established that should govern future sessions 4. Architecture diagrams — if a new component or flow was introduced 5. Sequence diagrams — if a new inter-component flow was added 6. Tech spec — if the implementation diverged from the written spec (the spec is the truth; update it) 7. Status.md — every session, without exception

The cascade is not supplementary to the work. It is the operation that closes the loop.

---

### 16. Context Loading Strategy

The order in which artifacts are loaded at session start determines what the AI holds as authoritative. Load in this order:

1. **Architectural constitution** (CLAUDE.md) first. This is the grammar. Everything else is read against it.

2. **Technical specification** — the section relevant to this session's scope. Load the whole document for cross-cutting tasks; load the relevant module spec for bounded tasks.
3. **Active ADRs** — those covering the area being modified. Do not load the full ADR directory on every session; load the decisions that govern the current scope.
4. **Status.md** — what was done, what is in progress, what is next.
5. **Session prompt** (bound roadmap item or improvised intent) — last, after all context artifacts are loaded.

**What to exclude.** Test files, unless the session is specifically about the test suite. Generated files. `node_modules`, `dist`, `build`. Lock files. Anything the AI would contextualize but not act on. Context budget is finite; loading noise competes with signal.

**MCP server budget.** A maximum of three active MCP servers in any session: the built-in file/search/terminal tools, optionally one semantic search tool (CodeSeeker) for large codebases, and optionally one specification-management tool (ForgeCraft). Each declared tool is read by the model on every turn whether invoked or not. Beyond three servers, tooling overhead begins to compete with the specification for context attention.

**The 300-line file read budget.** AI coding tools enforce a hard read limit per file invocation — approximately 2,000 lines / 25,000 tokens, after which content is silently truncated. Any file over 300 lines risks partial reads on any single invocation. The implication: the 300-line maximum on CLAUDE.md and spec files is not an aesthetic preference. It is calibrated to the tool's read budget. A 600-line constitution is not read in full; its lower half is invisible. This is the same constraint that makes the pointer architecture in §16 necessary: the spec is a skeleton with on-demand references, not a monolith the tool may or may not finish reading.

**Context compaction and specification artifacts.** When token pressure approaches the working limit, AI tools run an automatic compaction routine that retains a small number of recently active files and compresses all prior file reads and reasoning chains into a summary. Architectural decisions, constraint rationale, and prior session context are the first things lost. GS specification artifacts survive this because they are short, structured, and loaded at session start as priority context — they fit within the retained file budget compaction preserves. A README buried in session history does not. This is the structural reason GS artifacts outperform ad hoc documentation under sustained session pressure, not convention.

**Rationale: the token budget pressure that produced this pattern.** The pointer architecture described above was not designed top-down — it was forced by a concrete constraint encountered during ForgeCraft's own development. The original approach inlined all five GS procedure blocks (session loop, context loading, incremental cascade, bound roadmap, diagnostic checklist) directly into CLAUDE.md. Combined with three MCP servers' tool manifests and the session artifacts, the effective context window became mostly infrastructure and very little work. The model's attention degraded measurably: instructions near the bottom of a 400-line constitution were effectively invisible by turn 50.

The fix was architectural. CLAUDE.md was compressed to a skeleton with pointers (<200 lines). The detailed procedures moved to `reference.yaml`, fetched on-demand via `get_reference(resource: guidance)`. The procedures still exist at full fidelity, but they are loaded when *needed*, not on every turn. This in turn elevated the session prompt to the primary vehicle for task-specific working memory — because the constitution is deliberately minimal and the procedures are fetched rather than ambient. The bound prompt format (§5) emerged from this pressure: if the constitution is a skeleton, the prompt must carry everything the session needs.

The generative loop: token budget constraint → forced the constitution into a pointer document → forced procedures into on-demand reference → forced session prompts to be self-contained → produced the bound prompt format → which is now the Procedural Memory artifact the methodology recommends. The methodology produced the artifact format that the methodology needed. This is not circular — it is self-hosting, in the same sense that a compiler written in its own language is evidence that the language works.

---

## 17. Session Opening and Closing Rituals



**Opening checklist (30 seconds):** - [ ] Read Status.md — confirm the current state and the immediate next action - [ ] If returning from a waiting state of more than a few days: read the session summary - [ ] Confirm the pre-commit hook is active: `git status` and a small test commit should trigger it - [ ] Load the correct session-scoped prompt or clarify the intent for this session

**Closing checklist (5 minutes):** - [ ] Full test suite passes - [ ] Documentation cascade complete (§15 Phase 4) - [ ] Status.md updated with: what was done, current state, next action - [ ] Any decisions made that are not in an ADR: note them in Status.md “Decisions” section — these become ADR-write candidates - [ ] If the project goes to a waiting state: confirm the waiting-state record is complete enough that a cold start is possible from it alone

---

## Part VII: Operating Protocols

### 18. Initialization Protocol — New Greenfield Project

**Ceremony, step by step:**

1. Create the repository. Initialize git. Make the first commit: `.gitignore` only.
2. Run `setup_project` (ForgeCraft) with the project description. This generates the architectural constitution, selects relevant tags, and populates the initial `CLAUDE.md`. Review and customize.
3. Write the functional specification in `docs/specs/[project-name].md`. This is the human document; it does not need to cover the full system before step 4 — it needs to be precise and complete about the *current scope*. If you are applying the MVP entry path (§4), declare the scope boundary explicitly: what is in, what is deferred, and why.
4. Write the C4 context and container diagrams. Use a Mermaid block in the tech spec, or a dedicated diagram file. Commit.
5. Write initialization ADRs for every non-obvious decision made in steps 2–4.
6. Write use cases for the primary flows.
7. Write the bound roadmap: one prompt per major feature or subsystem in `DEVELOPMENT_PROMPTS.md`. Ordered by dependency. Each prompt self-contained.
8. Apply the derivability gate. Read the artifact set as if you were a stateless agent who had never seen the project. Can you derive what to build? If you would need to ask something that isn't written down, find the gap and fill it.
9. Create Status.md with the current state: initialization complete, next action is the first bound prompt.
10. Commit: `chore(init): initialization cascade complete – specification ready for implementation`.

The first implementation session begins from the first bound prompt. Not from a conversation. The specification is the conversation.

---

### 19. Brownfield Onboarding Protocol

A brownfield system is one that exists, works, and lacks any specification. The onboarding protocol does not rewrite the system. It writes the grammar the system should be governed by, then closes the gap between the existing code and that grammar, one atomic commit at a time.

**Phase 1 — Read the system.** Before writing a single specification artifact, spend one session reading: the entry points, the data models, the routing layer, any tests that exist. The goal is not to understand every file. It is to identify the system's natural layer structure, its domain entities, and its primary flows. Document what you find in rough notes, not in formal artifacts.

**Phase 2 — Write the specification for what it should become.** The architectural constitution is not a description of what the code does today. It is a specification of what it should do after the intervention. The gap

between today and that specification is the work. Write the constitution, the target architecture, and the ADRs for decisions you are making now. Commit the specification artifacts.

**Phase 3 — Install oracle tests.** Before any structural changes, write tests that cover the system's current observable behavior at the HTTP or CLI boundary. These are oracle tests: they document what the system does, not what it should do. Their purpose is to catch regressions introduced during the onboarding. They are not a permanent part of the test suite — they are replaced by proper unit and integration tests as the architecture is imposed.

**Phase 4 — Close the gap, one atomic commit at a time.** The brownfield cascade begins from the outermost layer inward: naming conventions first (rename, no behavior change), then layer extraction (extracting business logic from route handlers into service classes, for example), then test coverage (unit tests for extracted service methods), then error handling standardization, then the remaining hardening surface. Each commit is independently valid. No commit combines an extraction with a behavior change. The oracle tests run on every commit.

**Phase 5 — Retire oracle tests.** As proper unit and integration tests cover the extracted modules, oracle tests covering the same behavior are retired. The oracle test suite should be empty when the onboarding is complete.

## 20. Portfolio Management — The Cycling Model

A practitioner carrying multiple projects simultaneously is not context-switching between them in the session-execution sense. The model is cycling: at any given session, one or two projects are active (being executed); the others are in a waiting state (execution paused at a clean boundary).

**What a valid waiting state requires:** - The project's current commit passes the full test suite - Status.md names the immediate next action with enough specificity to begin a session from it - No partial work in the working tree — everything is either committed or explicitly staged with a note - Any inflight ADR decisions are noted in Status.md

A project in a valid waiting state requires zero execution from the practitioner until it is cycled back in. It does not require holding the project's context in working memory. The artifact set holds it.

**Cycling in.** When cycling a project back to active: 1. Read Status.md. One read should restore session-start position. 2. If the gap was long, read the session summary. 3. Run the test suite. Confirm the system is in the expected state. 4. Load the bound prompt for the next item. 5. Begin.

The portfolio size is bounded not by execution capacity but by specification bandwidth — the rate at which intent can be correctly externalized — and by how many projects can be maintained in a coherent waiting state. The second bound is softer than it seems. A project with a complete specification, a passing test suite, and a precise Status.md can wait indefinitely. The cost of cycling in is minimal. The cost of finding a project in an incoherent waiting state — partial work, no Status.md, tests failing — is full reconstitution from source.

## Part VIII: Test Architecture

### 21. Test Architecture by Layer

The test architecture is a first-class specification artifact. State it in the architectural constitution or a dedicated test architecture document. The AI generates tests from it; the agent defends regression with it; the commit pipeline enforces it.

**The harness as constitutive of the GS guarantee.** GS’s core claim — “the specification is the program” — holds only when the derivation is verified. The test harness is that verification. A specification without a harness is an assertion about intent; it is not a guarantee about behavior. An AI-generated codebase that is not continuously verified against the behavioral contracts in its specification may drift from those contracts silently — at generation speed, across sessions, without the practitioner noticing until integration. The T2 harness is not optional scaffolding that can be added later. It is structurally constitutive: removing it degrades the GS paradigm from a guarantee to a discipline preference.

In practice this means: the test architecture must be specified *before* any implementation session begins (it belongs in the initialization cascade, not the pre-release loop), and the gate conditions below are blocking acceptance criteria, not guidelines.

**Pipeline placement:**

Trigger	Test types that must pass
File save	Unit (affected module), lint
Commit	Unit (full suite), integration (affected service), type check, static security analysis
Pull request	E2E (full suite), visual regression, contract tests, accessibility gates, <b>mutation score gate (Stryker on changed modules, MSI ≥ 70%)</b>
Pre-deployment gate	<b>Full Stryker run on entire codebase (MSI ≥ 65%),</b> npm audit --audit-level=critical exits 0, full E2E on ephemeral env
Staging deploy	Smoke (APIs, UI, DB migrations, external deps), load tests (stated population + p99 ceiling), stress test to failure, DAST, penetration testing
Release candidate	All prior layers passed; chaos scenarios named in hardening spec; rollout parameters stated (canary population, error rate threshold, observation window)

**Coverage thresholds:**

Scope	Minimum
Overall	80% line coverage
New or changed code	90% (measured on diff)
Critical paths (data pipelines, auth, financial calculations, PHI handling)	95%+
<b>Mutation score (MSI) — overall</b>	<b>≥ 65%</b>
<b>Mutation score (MSI) — new or changed code</b>	<b>≥ 70%</b>

These are not aspirational. They are gate conditions. A commit that introduces new code below the threshold does not merge. The pre-commit hook enforces the overall threshold; the PR gate enforces the diff threshold.

**Why two separate thresholds?** Line coverage reports what was *executed*. Mutation score (MSI) reports what was *caught*. An AI-generated suite that exercises every line but asserts nothing produces 100% line coverage and 0% mutation score. The Shattered Stars project measured 80% line coverage before mutation testing; Stryker revealed 58% MSI. The gap — 22 percentage points — was entirely composed of tests that executed the right code but did not assert its behavior. All gaps resolved by adding targeted assertions, no new test infrastructure required.

## 22. GS-Specific Test Techniques

The following techniques are specific to generative specification practice and are not adequately covered by the standard testing literature.

**Expose-store-to-window.** For interactive applications (games, real-time UIs), expose the application state store to window in the test environment. Playwright assertions can then verify not only what the screen renders but what the application believes is true — the store's internal state — without coupling to DOM structure. This catches the class of failure that renders correctly but corrupts internal state: a score that displays right but is stored wrong.

```
// In test environment setup
if (process.env.NODE_ENV === 'test') {
  (window as any).__appStore = store;
}

// In Playwright test
const score = await page.evaluate(() => window.__appStore.getState().player.score);
expect(score).toBe(expectedScore);
```

**The vertical chain test.** A single UI action triggers Playwright, which then queries the service layer response, the database state, and any affected indexes — verifying correct propagation through every boundary the action crosses — then returns to the UI to confirm the visible outcome matches the stored state. One trigger, inspected at every boundary it crosses. The test specification names which critical flows receive this treatment.

**Mutation testing as adversarial audit.** An AI-generated test suite carries a structural risk: tests written by a system that knows the correct implementation may be written to pass it rather than to catch violations of it. Mutation testing closes this gap by introducing deliberate behavioral faults — inverting a condition, replacing an operator, removing a return value — and verifying that the suite detects each one. A test that passes a mutant is not testing the contract. Coverage measures what was executed. Mutation score measures what was caught.

This is not a pre-release activity. **Run Stryker immediately after writing any test batch**, before moving to the next module. A Stryker run on a single module takes seconds. A Stryker run on an untested codebase discovered at release takes hours and surfaces test rewrites across many modules simultaneously. Catch the gap at the moment of creation.

Mandatory cadence: 1. Write tests for a module. 2. Run `npx stryker run --mutate src/[module]/**` — targeted to that module only. 3. Surviving mutants identify assertions that are missing. Add them before proceeding. 4. PR gate runs Stryker on all changed modules automatically. 5. Release candidate runs Stryker on the full codebase as a final gate.

Mutation score thresholds are in §21 (coverage thresholds table). They are gate conditions, not guidelines.

Recommended tooling: stryker-mutator (JavaScript/TypeScript), mutmut (Python), Pitest (Java).

**Multimodal quality gates.** For generative assets (images, audio), define acceptance criteria as executable validation:

- *Visual:* PCA on the sprite silhouette to extract the primary axis; assert angle within tolerance bounds. Standard math library (numpy, scikit-learn) applied as a domain-specific gate. Constraints are in the specification; the tool is already on the shelf.
- *Audio:* Tempo consistency, frequency profile (no asset competing in the 2–4 kHz presence range during dialogue), loudness normalization to target LUFS, silence detection. Assertable from audio analysis libraries against each generated output before it reaches the runtime bundle.
- *MCP-mediated inspection:* An instrumented application state exposed through an MCP server, accessible to the AI during a test session. The model receives a scene description and acceptance criteria, loads live

### 23. The Hardening Surface

Functional tests verify that the system does what it should. Hardening tests verify that the system does *only* what it should, and that it survives adversarial and operational conditions. The specification of hardening tests is part of the test architecture document — not an afterthought added before release.

Category	Constraint vocabulary	Representative tooling
Stress & performance	Peak concurrent users, sustained request rate, p99 latency ceiling, error rate threshold; soak, spike, and ceiling variants	k6, Artillery, Locust
Security	Authentication bypass, injection payloads, dependency CVE scan, CORS policy, secret exposure, privilege escalation; severity acceptability threshold	npm audit, Snyk, OWASP ZAP
Chaos engineering	Recovery time after node kill, dead-letter injection, DB failover window, circuit breaker thresholds	Chaos Monkey, Gremlin, custom fault injectors
Cross-cutting concerns	Encryption policy, authorization model (RBAC/ABAC per surface), observability schema (correlation ID, PII redaction, SLO thresholds), data lineage contract, dependency license compliance	TLS auditors, log schema validators, audit tooling
Environment hardening	TLS headers, Content Security Policy, IAM least-privilege boundaries, CORS policy correctness — agent audits running environment against spec and closes the delta	Trivy, tfsec, cloud provider policy tools

The common failure pattern: hardening requirements fail not because engineers are unaware of them, but because they were never stated as blocking acceptance criteria. The specification makes them structurally present — explicit, enforced, verifiable. An agent that can reach the CLI can run every category above; the specification tells it what passing means.

**Dependency governance as a prescriptive specification requirement.** The GS experiment series demonstrated that architectural correctness (passing the GS audit) and dependency security are fully orthogonal. A project that scores 12/12 on the GS rubric — perfect layer discipline, full enforcement infrastructure, complete audit trail — can simultaneously carry nine high-severity CVEs from an unconstrained devdependency chain. The rubric does not measure what the AI selected; it measures how the AI structured what it produced. Both axes of quality must be specified explicitly, because the AI will not apply safety constraints that are not stated.

Prescriptive dependency governance belongs in the architectural constitution, not left to model discretion. The minimum specification:

- Dependency selection (non-negotiable):
- Password hashing: argon2 or bcryptjs. Not bcrypt: its native binding pulls

- @mapbox/node-pre-gyp → tar, a known CVE chain.
- npm audit gate: zero HIGH vulnerabilities required as a P1 acceptance condition.

- devDependencies: verify that typed linting packages do not introduce transitive CVEs via old minimatch versions before committing.

The multi-agent experiment’s dependency governance condition (GS v3) confirmed that a single specification directive of this kind eliminates the entire high-CVE surface across all relevant chains. Add this block to the CLAUDE.md zero-hardcoded-values section or a standalone DependencyPolicy.md artifact in the GS cascade. Any project where the model selects dependencies without explicit direction is operating with an unconstrained executor in the supply-chain dimension.

## Part IX: Toolchain

### 24. ForgeCraft — Specification Scaffolding

ForgeCraft (forgecraft-mcp@1.5.0) generates production-grade architectural constitutions from a library of 116 curated template blocks covering 24 project classification tags and six AI assistants. The 1.4.0 release added five-phase quality gates (T1–T4 cascade enforcement), ADR sequencing, live documentation hooks, and guided practitioner feedback at each phase close. The 1.5.0 release added agent-agnostic session advising, pre-implementation impact assessment, spec consistency scanning, and postcondition coverage scoring.

**Install:** npx forgecraft-mcp@1.5.0

**CodeSeeker is bundled by default.** ForgeCraft 1.5.0 includes CodeSeeker as a recommended companion. CodeSeeker provides graph-based code intelligence (imports, calls, extends) that fills the gap grep cannot: re-exports, dynamic imports, type references vs value references, and barrel file entries. The rationale is structural — grep is text pattern matching, not an AST; any rename or interface change that relies on grep alone will miss these cases. Projects initialized with ForgeCraft get the CodeSeeker recommendation automatically.

**When to run:**

Command	When
setup_project	New project or complete specification rebuild
refresh_project	Scope has drifted (new framework, new tag category) — detects drift and regenerates cleanly
advise_session	Session start — reads project signals and returns a prioritised advisor block. Works on any project; no forgecraft.yaml required. Install the companion session-advisor.sh UserPromptSubmit hook to inject state automatically before every prompt.
propose_session	Before starting implementation — runs pre-implementation impact assessment, produces proposal.md with spec delta, layer readiness per UC, and open clarifications
check_spec_consistency	Before a major feature or release — scans all spec artifacts for gaps, orphan probes, hollow probes, stale ADRs, and unresolved [NEEDS CLARIFICATION] markers
audit_project	Before a major release or external review — scores compliance and identifies gaps
review_project	Pre-merge review — structured checklist across architecture, quality, tests, performance

**Command****When**

scaffold\_project

Generate folder structure, hook skeletons, and documentation scaffolding for a new module

**Tag selection.** Tags activate domain-specific standards. UNIVERSAL applies to all projects. Common additions: - API — adds REST/GraphQL constraints, API versioning discipline, contract testing - WEB-REACT — adds component architecture rules, state management constraints, accessibility gates - DATA-PIPELINE — adds data lineage requirements, schema validation, idempotency constraints - CLI — adds exit code standards, stdin/stdout handling rules, non-interactive mode requirements

**Tier selection:** - core — essentials only; use for projects with tight context budget or to compress an overgrown constitution - recommended (default) — core + best practices; appropriate for most projects - optional — everything including advanced patterns; use when the advanced patterns are actually needed **GS ecosystem compounding.** The AX v6 experiment (§S9.7 of the Experiment Supplement) confirmed that GS-built tooling measurably improves GS-guided builds. Activating CodeSeeker v2.0.0 during generation reduced structural duplication from 5.37% to 2.50% and eliminated interface completeness gaps that static prompting missed. The principle generalizes: each GS-built tool that enters the ecosystem becomes a quality gate available to the next project. The flywheel is real and measurable.

## 25. The Infrastructure Execution Model

An AI with CLI access is an executor, not an advisor. Cloud infrastructure is not a separate discipline requiring separate tooling expertise. It is another surface the AI executes against, given a specification of the desired infrastructure state.

### What the specification must state for infrastructure:

- Desired resource topology (services, their sizes, their regions)
- IAM boundaries (what may call what, with what permissions)
- Encryption policy (at rest, in transit, TLS version)
- Ingress/egress rules
- Monitoring requirements (metrics, alerts, dashboards)
- Tagging policy (cost attribution, environment labeling)

With these stated, the AI can provision, wire, validate, and iterate without returning to the engineer between commands. The Shattered Stars environment configuration case, the SafetyCorePro data warehouse module, and the BRAD RAPTOR indexing infrastructure were all executed at the CLI level — not proposed for manual execution.

The scope of what must be specified extends beyond code to every domain the agent touches. This is the direct consequence of the CLI execution model: the specification becomes the operational grammar for an agent that can act.

## Part X: GS Beyond Application Code

The seven properties of a generative specification have no ceiling at the code layer. The constraint vocabulary changes by domain. The mechanism — desired state, blocking acceptance criteria, automatic rejection of non-conforming output — does not.

## 26. Generative Asset Pipelines

An AI-generated asset — a sprite sheet, a sound effect, a music track — is valid or invalid relative to a specification, exactly as a TypeScript module is valid or invalid relative to an interface. The difference is that the specification for a visual asset is expressed as measurable acceptance criteria on the asset’s properties.

**Visual assets.** The acceptance criteria must be computable from the asset file itself:

Constraint	Mechanism	Example threshold
Vertical symmetry	Compare pixel-level left/right halves after horizontal flip; normalized similarity	$\geq 0.85$
Orientation	PCA on pixel mass; assert principal axis angle from vertical	$\leq 15^\circ$
Background cleanliness	Non-background pixel ratio in border region	$\leq 0.30$
Style consistency	Multimodal model evaluation against style specification	Structured deviation report

A sprite that fails any check is rejected and regenerated automatically — no human review at scale. The symmetry threshold is a named constant in the specification; the validator is written against it; conforming output is selected without manual intervention.

**Multimodal QA for existing libraries.** Submit each asset to a vision model with the visual specification as the evaluation rubric. The model returns structured identification of violations — incorrect orientation, palette deviations, style inconsistencies — at the scale and speed manual review cannot match.

**Audio assets.** Computable gates for generated audio: - Tempo consistency within scenes - Frequency profile compliance (no asset should occupy the 2–4 kHz presence range during dialogue) - Loudness normalization to a target LUFS value - Silence detection for generation artifacts

These are assertions against audio analysis libraries. The pipeline structure is identical to a unit test suite: each check has a threshold, each threshold is named in the specification, each failure triggers regeneration.

## 27. Infrastructure as Desired-State Specification

An AI with CLI access is an executor, not an advisor. Cloud infrastructure provisioning follows the same grammar as application code: desired state, acceptance criteria, agent iteration to close the gap.

**What the infrastructure specification must state:**

- Resource topology (services, sizes, regions)
- IAM boundaries (explicit allow-list: what may call what, with what permissions)
- Encryption policy (at rest, in transit, TLS version, rotation schedule)
- Ingress/egress rules and CORS policy
- Monitoring requirements (metrics, alert thresholds, dashboards)
- Cost tagging policy (environment label, team attribution, project attribution)

With these stated, the AI issues every command: provisioning, IAM wiring, VPC configuration, stack deployment, and validation. A full ETL pipeline — data ingestion, transformation, storage, monitoring, alerting, and all non-functional requirements — can be built at the CLI level without the engineer issuing platform-specific commands.

The acceptance criteria have the same structure as application tests: the deployment is valid only when the running environment satisfies every stated constraint. Missing resource, wrong policy, misconfigured certificate — each is a failing assertion, not a configuration detail for the engineer to notice.



## 28. The Business Layer

An AI generating marketing strategy, pricing decisions, content calendars, or competitive positioning without stated acceptance criteria produces output that is fluent, confident, and potentially wrong in ways the domain makes invisible without explicit constraints.

The specification has two axes at the business layer:

**Economic viability constraints.** State the acceptance criteria before instructing the AI to produce strategy output: - Target conversion rate for content - Sustainable publishing cadence (audience retention threshold, not platform-maximum cadence) - Cost-per-acquisition ceiling - Revenue floor before a channel is worth maintaining

An AI generating content strategy without these will optimize for something — usually volume or engagement signals — not for the constraints that govern business survival.

**Legal and ethical compliance constraints.** Jurisdictional requirements, regulatory constraints, data rights, contractual obligations, and ethical commitments are not soft preferences. They are the constraints that define what counts as a valid output in the domain of consequential decisions. A pricing strategy that crosses into collusion, a piece that misrepresents a product, or a communication that violates a contributor’s rights has passed no stated acceptance criterion — because none was stated. The failure is structural.

The test is the same as at the code layer: the constraint must be blocking and automatic, not advisory. A content calendar with a stated audience retention threshold and a validator that flags output below it is a production rule. A content calendar with a “brand voice” section and no enforcement mechanism is a README.

---

## 29. Model Selection — A Practitioner Note

*This section reflects the author’s practice as of March 2026. Model capabilities shift faster than any document should claim permanence. Treat this as a current baseline and experiment beyond it.*

The multi-agent experiment series in the companion white paper used **claude-sonnet-4-5** throughout. The personal inflection point came with **claude-opus-4-5**: the first model where GS artifact fidelity — reading the full CLAUDE.md, honoring all layer rules across a six-prompt session, emitting infrastructure artifacts on explicit directive — became reliably reproducible rather than session-dependent luck. What followed was unexpected: **claude-sonnet-4-6**, a smaller and cheaper posterior model, performs at or above the opus-4-5 level for GS-governed coding tasks. Fewer parameters, later training, and apparently a compression of what opus demonstrated into a model that costs a fraction of it.

### Current recommendation:

Use case	Model	Rationale
Primary specification work, complex sessions	claude-sonnet-4-6	Later training than opus-4-5; at or above opus performance for GS coding tasks at significantly lower cost; preferred default
Maximum reasoning depth, novel domains	claude-opus-4-5	The inflection-point model; reserve for sessions where the problem is genuinely open-ended or the specification is being written for the first time

Use case	Model	Rationale
Cost-sensitive coding tasks, high-volume generation	claude-sonnet-4-5	Strong GS compliance; appropriate when the task is well-bounded and the specification is mature
Experimental / adversarial audit	Any	Use a fresh session with no added context for blind audit runs — model version is less important than isolation

**What to look for in a model upgrade.** The properties that matter for GS practice, in order of importance: 1. *Instruction-following fidelity at depth.* Does the model honor a rule stated on line 140 of a 200-line CLAUDE.md by turn 8 of a session, without the practitioner repeating it? 2. *Emit discipline.* Does the model emit fenced file blocks rather than prose descriptions when explicitly directed to? 3. *Layer discipline unprompted.* Does the model maintain architectural boundaries (no repository calls in route handlers, no Prisma in services) without per-prompt reminders? 4. *Coverage honesty.* Does the model report measured numbers rather than aspirational ones in generated documentation?

These are testable. Run the blind adversarial audit from the companion supplement against a new model on a known benchmark before committing to it for production sessions.

**This will change — and the methodology improves with it.** Model capability and GS practice are complementary, not competing. Every generation that improves instruction-following fidelity, emit discipline, or architectural reasoning makes a complete specification more productive: a better reader executes the same grammar more faithfully. Some of the most explicit directives in current templates — emit this file in P1, do not leave this field as TBD, name the files you reference — exist because today’s models require that level of precision. As models improve, that surface area shrinks. A directive necessary at sonnet-4-5 may be redundant at whatever comes next. That is not the methodology becoming obsolete — it is the compliance scaffolding thinning as the reader requires less of it. The core does not thin: architectural decisions, domain contracts, behavioral boundaries, decision rationales. Those are system-level artifacts; a model that never forgets still needs to be told what the system is.

The right practice is periodic re-evaluation against a fixed benchmark — not loyalty to a named model. Run the blind adversarial audit from the companion supplement against any new model on a known benchmark before switching. What changed and in which direction is the question; the answer updates the practitioner’s infrastructure, not their methodology.

### 30. Change Governance by Construction

Enterprise change management frameworks — ITIL, ITSM, COBIT — require that changes to production systems be initiated through a formal request, reviewed by a change advisory body, traceable to a decision record, and auditable after the fact. Teams that adopt AI-assisted development typically face a version of this objection: *if the AI generates code, who approved the change? Where is the audit trail?*

GS resolves this structurally rather than procedurally. The artifact grammar satisfies the change governance requirement by construction. There is no separate documentation step because the governance artifacts *are* the development artifacts:

ITIL/ITSM concept	GS artifact	Location
Request for Change (RFC)	ADR — the decision record that precedes any architecture-level change	docs/decisions/ADR-NNN.md

ITIL/ITSM concept	GS artifact	Location
Change Advisory Board (CAB) review	ForgeCraft gate stack — the gate conditions that must pass before a merge	Enforced at PR time via quality gates
Change record	Commit log (conventional commits, scoped, typed)	Git history — every merge is a typed, scoped record
Audit trail	ADR lineage + git log + Status.md	Queryable from version control; no separate system required
Post-implementation review	Documentation cascade closure — artifacts updated after every increment	Session close protocol (§17)
Configuration item (CI)	Every named artifact in the GS cascade — spec, ADRs, diagrams, constitution	Version-controlled alongside code

The practical consequence for regulated environments (HIPAA, SOC 2, CMS, PCI-DSS): the compliance artifact is not produced separately from the build artifact. It is the same artifact, read from two directions. An auditor asking “what changed, when, and who approved it?” receives the answer from `git log` and the ADR index. An engineer asking “what should I build next?” receives the answer from the same ADR index and the current specification.

COMPASS (the multi-tier regulated data platform case study in the companion white paper) demonstrates this at scale: the specification that governs the ETL architecture is simultaneously the change control record for every data flow, schema contract, and monitoring threshold in the system. The lineage graph — T1 spec → T2 harness → T3 infrastructure → T4 monitoring — is the audit-ready change history. Adding a new data source requires an ADR (the RFC), passes ForgeCraft gates (the CAB), lands in the git log (the change record), and updates Status.md (the post-implementation review). The process is identical for one practitioner on a greenfield project. The formality is inherent in the methodology, not added by the organization.

**What this means in practice:** When adopting GS in an organization with existing change management processes, map the GS artifact grammar onto the ITSM vocabulary before writing the first spec. Identify which artifact plays which ITSM role. Resistance from process owners almost always dissolves at this mapping — not because GS bypasses governance, but because it makes governance inseparable from the act of building.

## Appendix A: Artifact Quick Reference

Artifact	Memory type	Frequency	Owner	Gate-critical
CLAUDE.md	Semantic	Update on architecture change	Practitioner (generated by ForgeCraft)	Yes — read every session
Corrections Log (in CLAUDE.md)	Semantic	Any session with a corrected AI pattern	Practitioner + AI	No — accumulates; prevents pattern recurrence
Techniques subsection (in CLAUDE.md)	Semantic	When a named technique is adopted	Practitioner	No — activation registry
Tech spec	Semantic	Update when behavior changes	Practitioner	Yes — read at implementation
ADR	Episodic	Every non-obvious decision	Practitioner	No — consulted on conflict

4/21/26, 3:51 PM

Generative Specification: The Practitioner's Protocol

Artifact	Memory type	Frequency	Owner	Gate-critical
Status.md	Episodic	Every session close	Practitioner	Yes — required for session close
Session summary	Episodic	Long waiting states	Agent	No — consulted on cold start
C4 diagrams	Relationship	New component or boundary	Agent (from spec)	Yes — precedes implementation
Sequence diagrams	Relationship	New inter-component flow	Agent (from spec)	Yes — precedes implementation
State machines	Relationship	Modal behavior surfaces	Agent (from spec)	Yes — precedes implementation
Use cases	Relationship	New behavioral contract	Agent (from spec)	Yes — triple derivation source
DEVELOPMENT_PROMPTS.md	Procedural	Before implementation of each item	Practitioner	Yes — required for waiting state
Session prompt	Working	Per session	Practitioner	Yes — session start artifact
Loaded context	Working	Per session	Agent	Yes — ordered load sequence

---

## Appendix B: The Memory Taxonomy as Diagnostic Checklist

Before beginning any session on a new or inherited project, run this check:

- SEMANTIC MEMORY
- [ ] CLAUDE.md exists and is current (not describing a prior architecture)
  - [ ] Technical specification covers current scope
  - [ ] Domain vocabulary is defined (no ambiguous names)
- PROCEDURAL MEMORY
- [ ] Bound prompts exist for the next 3+ roadmap items
  - [ ] CI/CD pipeline is specified (not just configured)
  - [ ] Pre-commit hooks are active and tested
- EPISODIC MEMORY
- [ ] ADRs exist for every non-obvious architectural decision currently in force
  - [ ] Status.md was updated at the close of the last session
  - [ ] Git history is typed (conventional commits, not 'wip' and 'changes')
- RELATIONSHIP MEMORY
- [ ] C4 L1 and L2 diagrams are current
  - [ ] Primary flows have sequence diagrams
  - [ ] Use cases cover every behavioral contract being modified this session
- WORKING MEMORY
- [ ] Session prompt (or intent) is specific enough to begin without narration
  - [ ] Context loading plan covers what's needed and excludes what isn't
  - [ ] MCP server count  $\leq 3$

A “no” on any semantic or procedural item is a blocker. Fill the gap before beginning implementation. A “no” on episodic, relationship, or working items should be resolved in the first 10 minutes of the session.

---

# Appendix A: Bound Prompt Exemplars

The bound prompt format defined in §5 is a template. This appendix provides three worked examples from a real production project (ForgeCraft MCP), each representing a different task shape. The annotations explain *why* each section exists, mapping the prompt structure back to the GS properties and the five memory types.

## Reading the Annotations

Each prompt section serves a specific structural function:

Prompt Section	GS Property	Memory Type	What it prevents
Specification references	Bounded	Working	Agent loads noise instead of signal
Precondition	Verifiable	Episodic	Task starts from invalid state
Scope	Bounded	Semantic	Scope creep within session
Acceptance criteria	Verifiable	Procedural	“Done” is subjective
Architecture constraints	Defended	Semantic	Agent violates layer rules
Commit message	Auditable	Episodic	Git history is untyped

## Exemplar 1: Documentation Task (No Code Changes)

**Task shape:** Pure specification work — writing a new section of a practitioner document. No implementation code changes. Tests this pattern: can a bound prompt drive documentation with the same rigor as implementation?

## P-001 – Add §16 Context Loading Strategy to Practitioner Protocol

```
**Specification references:**
- Load `CLAUDE.md` (architectural constitution)
- Load `docs/forgecraft-spec.md` §4 (GS methodology)
- Load `Status.md` (current session state)
- Load `templates/universal/reference.yaml` (guidance blocks for exact wording)
- Do NOT load test files, `node_modules`, or `dist`

**Precondition:**
The practitioner protocol white paper exists in its working location. The five GS
guidance procedure blocks have been moved from `instructions.yaml` to
`templates/universal/reference.yaml` with `topic: guidance`. The `get_reference`
tool is implemented and callable.

**Scope:**
- ADD: New §16 "Context Loading Strategy and On-Demand Procedure Dispatch"
- ADD: Description of the `get_reference(resource: guidance)` tool contract
- ADD: Rationale for the token-budget constraint motivating the pointer pattern
- UPDATE: Any existing section referencing inlining GS procedures into CLAUDE.md
- NOT IN SCOPE: Changes to §1-§15

**Acceptance criteria:**
- [ ] §16 describes context loading order (constitution → spec → ADRs → Status.md → prompt)
- [ ] §16 covers the CLAUDE.md token-budget constraint (<200 lines target)
- [ ] §16 covers `DEVELOPMENT_PROMPTS.md` as the Procedural Memory artifact
- [ ] §16 explains the MCP server budget limit (≤3 active servers)
- [ ] Section numbering is consistent after addition
```

**\*\*Architecture constraints:\*\***

- Documentation task only – no code changes
- Write in the white paper's existing register (principled, practitioner-oriented)
- Each paragraph must be independently purposeful – no filler transitions

**\*\*Commit message:\*\*** docs(white-paper): add §16 context loading strategy

**Why this exemplar matters.** Documentation is where most teams abandon rigor. A bound prompt for a documentation task demonstrates that the Bounded and Verifiable properties apply to every artifact type, not just code. The “NOT IN SCOPE” line is the most important line in the prompt — without it, the agent will “improve” existing sections while writing the new one.

---

## Exemplar 2: Verification and Fix Task (Existing Code, Tests Only)

**Task shape:** Coverage gap — existing implementation code is correct but undertested. The agent must fix test files without modifying application code. Tests this pattern: can a bound prompt constrain the agent to a read-only posture toward production code?

**## P-002 – Artifact Coverage: Fix 0% core + 37% artifacts Coverage**

**\*\*Specification references:\*\***

- Load ``CLAUDE.md`` (testing pyramid section)
- Load ``src/core/index.ts`` (GenerativeSpec type exports)
- Load ``src/artifacts/index.ts`` (public artifact barrel)
- Load ``tests/core/properties.test.ts`` (pattern reference)

**\*\*Precondition:\*\***

``tests/core/properties.test.ts`` exists. Five test files in ``tests/artifacts/`` have been created. Coverage baseline: ``src/core`` 0%, ``src/artifacts`` 37%.

**\*\*Scope:\*\***

- VERIFY: ``npm test -- --coverage`` runs cleanly
- FIX: Import path errors, missing exports, or wrong constructor signatures in tests
- TARGET: ``src/core`` ≥ 80%, ``src/artifacts`` ≥ 80%
- NOT IN SCOPE: Changing artifact implementation code. Fix tests only.

**\*\*Acceptance criteria:\*\***

- [ ] ``npm test`` exits 0 with all tests passing
- [ ] ``src/core`` coverage ≥ 80%
- [ ] ``src/artifacts`` coverage ≥ 80%
- [ ] No skipped or pending tests added

**\*\*Architecture constraints:\*\***

- Test against public API only – no testing of private methods
- Use ``mkdtempSync`` for any tests requiring real filesystem interaction

**\*\*Commit message:\*\*** test(artifacts): fix coverage – src/core and src/artifacts gates

**Why this exemplar matters.** The critical constraint is “NOT IN SCOPE: Changing artifact implementation code.” Without this line, an agent facing a test failure will fix the production code to make the test pass — the path of least resistance. The bound prompt makes that path architecturally unreachable by declaring it out of scope. This is the Defended property applied to a session prompt, not a commit hook.

---

## Exemplar 3: Additive Implementation Task (New Tests, Integration Boundary)

**Task shape:** New integration test for an existing tool. The agent adds test cases that exercise a real handler against real data. Tests this pattern: can a bound prompt drive integration-level work with precise verification?

### ## P-003 – Add `get_reference(guidance)` Integration Test

**\*\*Specification references:\*\***

- Load ``src/tools/get-reference.ts`` (`getGuidanceHandler`)
- Load ``tests/tools/get-reference.test.ts`` (existing test patterns)
- Load ``templates/universal/reference.yaml`` (guidance blocks)

**\*\*Precondition:\*\***

``getGuidanceHandler`` is implemented and exported. The five guidance blocks exist in ``reference.yaml`` with ``topic: guidance``. The router dispatches ``resource: "guidance"`` to ``getGuidanceHandler()``.

**\*\*Scope:\*\***

- ADD: Integration test verifying `getGuidanceHandler()` returns 5 guidance blocks
- ADD: Content assertions for "Session Loop" and "Context Loading"
- ADD: Exclusion assertion – guidance blocks NOT in composed instruction blocks

**\*\*Acceptance criteria:\*\***

- [ ] ``npm test`` exits 0 with new tests passing
- [ ] At least 3 test cases for ``getGuidanceHandler``
- [ ] No guidance block IDs in ``composed.instructionBlocks`` when composing

**\*\*Commit message:\*\*** test(get-reference): add integration tests for guidance resource

**Why this exemplar matters.** The third acceptance criterion is a *negative assertion* — verifying that something does NOT happen. An unbound prompt (“write tests for the guidance handler”) would never produce this. The practitioner who wrote the prompt knows that the guidance blocks were intentionally excluded from composed instruction output, and that this exclusion must be verified. Domain knowledge flows into the prompt; the agent executes the verification. This is the division of labor the methodology depends on.

## The Pattern Across All Three

Every exemplar follows the same structure, but each exercises a different constraint:

1. **P-001** constrains *what the agent writes* (documentation, not code)
2. **P-002** constrains *what the agent touches* (tests only, not production code)
3. **P-003** constrains *what the agent verifies* (including a negative assertion the agent would never generate unprompted)

The practitioner’s skill is not in the format — the format is a template. The skill is in knowing which constraint to impose for each task shape. That knowledge is domain expertise expressed as specification structure.

*This document is maintained as the companion execution guide to the Generative Specification white paper. When this document and the white paper diverge, the white paper holds the theoretical argument; this document holds the current operational protocol. Revise this document when the protocol changes; revise the white paper when the paradigm argument changes.*

1. The five-category taxonomy adapts Tulving’s multiple memory systems theory (Tulving, 1972; 1985), which formally distinguished **episodic memory** (personally experienced, time-indexed events) from **semantic memory** (general world knowledge independent of acquisition context). The episodic/semantic distinction maps directly onto the **Episodic** and **Semantic** artifact classes above. The **Procedural** category

draws on the declarative/procedural memory distinction formalized by Squire (1987) — procedural memory encodes *how* to do things implicitly, without conscious access to the rule; GS procedural artifacts make that implicit rule explicit so a stateless reader can access it. The **Relationship** and **Working** memory categories are adaptations specific to the stateless-reader context; they are not standard cognitive science categories, though Working Memory as a distinct system is well established (Baddeley, 1974). The taxonomy is applied here instrumentally — as a diagnostic grid for artifact coverage — not as a claim about the cognitive architecture of LLMs.↩