

Generative Specification: A Pragmatic Programming Paradigm for the Stateless Reader

Author: Juan Carlos Ghiringhelli (Pragmaworks)

Version: 2.0

Date: April 2026 **Status:** Preprint

Prologue: The \$327 Million Contract That Was Never Written

On September 23, 1999, the Mars Climate Orbiter completed a nine-month, 416-million-mile crossing of interplanetary space. It arrived within 26 kilometers of its intended trajectory — a feat of precision that represented the combined work of thousands of engineers, two years of mission planning, and \$327.6 million in public investment. Then it entered the Martian atmosphere at the wrong angle and was destroyed in 57 seconds.

The cause was not a bug in any individual system. Lockheed Martin's navigation software reported thruster force in pound-force seconds. NASA's flight computer expected newton-seconds. Both teams had implemented their components correctly, according to their own assumptions. The interface between the two systems had no explicit unit specification. The contract had been assumed, not written. *(NASA MCO Mishap Investigation Board, Final Report, November 1999; analysis confirmed in the Columbia Accident Investigation Board framework as a category of organizational interface failure.)*

No test caught it. The code compiled cleanly. Individual modules passed validation. The failure was invisible everywhere except at the boundary — at the seam where two systems, each internally coherent, had to agree on a shared language. They did not agree, because the agreement had never been formalized.

That was 1999. That contract failure took two years and \$327.6 million to produce.

Today, in current practice, an AI assistant produces an equivalent interface in thirty seconds (author's observation). It generates clean code. Types check. Unit tests pass. And embedded in that output, invisible, is a set of implicit assumptions — about units, about field ordering, about what a null means, about which layer owns which concern — that the generating model resolved silently, drawing on everything it has ever read about how such systems are usually built. The code is correct in isolation. At the boundary, when two AI-generated systems meet across a session boundary or a team boundary or a service boundary, the implicit assumptions do not automatically agree.

The Orbiter problem did not go away. The velocity of producing it increased by roughly two orders of magnitude (author's estimate).

This is not an argument against AI-assisted development. It is an argument for what that development requires. The Jacquard loom, introduced in 1804, did not eliminate weaving craft — it relocated it. The weaver stopped managing the shuttle. The weaver started managing the card. The complexity moved upstream, into the specification, where it could be expressed once and executed at machine speed. The loom was capable. The card made that capability directed.

Software engineering is undergoing the same relocation, at a speed the loom's inventors could not have imagined. An AI agent with CLI access can read your codebase, write tests, execute migrations, commit to git, and iterate on a running system, all within a single session, starting from nothing. The executor is extraordinarily capable. The question is what governs it across the session boundaries it cannot see, the team context it was never given, and the architectural decisions that were made in conversations it was not part of.

The answer is the same as it was in 1804. The answer is the same as it was in 1999. A specification precise enough that a stateless reader — one with no memory of your intentions, no shared context, no ability to ask a clarifying question — can derive correct output from it alone.

That discipline is what this paper defines.

This paper requires an AI agent with direct CLI access — not a chat assistant, but an agent that can read files, write files, run tests, commit to git, and execute commands. Claude Code, Cursor, and VS Code agents in agentic mode meet this requirement. The methodology is not applicable to chat-based interfaces. §8.6 develops the runtime requirements in full.

Abstract

The dominant failure mode of AI-assisted development is not incorrect code. It is **architectural drift**: structurally incoherent output produced at generation speed across sessions that share no persistent context. Each session starts stateless — no memory of prior decisions, no institutional context, no accumulated conventions. Without an explicit specification, intent degrades at every session boundary. The Orbiter problem did not go away. The velocity of producing it increased by roughly two orders of magnitude.

Generative Specification (GS) is the programming discipline of the pragmatic tier: the tier at which *derivability* — what a stateless reader can correctly determine from artifacts alone — becomes a binding constraint. It names the obligation that AI-assisted development makes structurally necessary, in the sense Robert C. Martin used when characterizing structured programming, OOP, and functional programming: a discipline defined by what it removes from programmer freedom. What GS removes is implicit context.

Seven specification properties operationalize the discipline — **Self-describing, Bounded, Verifiable, Defended, Auditable, Composable, Executable** — each named for a specific failure mode observed across six production projects. The obligation is direct: make each class of failure structurally unreachable. The economic consequence of the structure has a name: **cost inversion** (§4.1.b). When regeneration is near-free, specification becomes the scarce resource.

Empirical evidence spans six production projects, a controlled adversarial study reaching 14/14 on the rubric with 109 passing tests, a replication experiment reproducible by any reader with an API key, and a human practitioner study (58 developers, April 2026, two-project crossover design). The DX1 finding is two-part and honest about both parts: GS discipline transfers in one session, and ForgeCraft's artifact generation revealed a temporal mismatch under time pressure whose fix is a product change, not a methodology change. A follow-up transfer study (DX2, May 2026) is in progress.

The full argument, evidence, and implications follow. The discipline is replicable: §7.4 (RX) reproduces a scoped implementation from a fresh GS document with 104 passing tests, starting from nothing.

The specification is the mold. The AI is the foundry.

1. Introduction

The dominant failure mode of AI-assisted development is architectural drift. This paper names its structural cause, defines the discipline that resolves it, and provides evidence across six production projects, a controlled adversarial study, and a 58-participant practitioner experiment.

Hoare logic (1969), design by contract (Meyer, 1992), REST (Fielding, 2000), and the semantic web — every one of these formal disciplines for correct computing was provably right. None was fully implemented at scale. The formal theories were complete. The executor was not.

The missing executor was not intelligent — it was *consistent*. These disciplines failed not because their proofs were wrong but because sustaining them required what humans reliably cannot provide across every person, every commit, every deadline. That executor has changed.

Programming languages were never designed for you to think in. They were designed as messengers — protocols for translating human intent into what a context-free parser could deterministically read. The new reader — a large language model — is not a context-free parser. Trained on the full corpus of human-written text and code, it interprets each token relative to everything surrounding it in the context window: the import statements, the class hierarchy, the architectural rules in the session-loaded documentation. The constraint that made programming languages rigid is no longer operative. The specification can now operate at a higher abstraction level than the implementation language, stated in the language of the domain, derived into implementation by a reader that understands the surrounding context.

But this requires discipline. The new reader is stateless: no memory of prior sessions, no institutional context, no tolerance for the implicit. Incomplete descriptions are completed at generation speed, consistently, incorrectly. Architectural drift is the observable result — multi-session accumulation of locally valid but architecturally incoherent output. The cure must persist across session boundaries.

The most common objection — that better prompting solves drift — is addressed in §8.9. A prompt is a session artifact. The cure for a multi-session problem cannot be a single-session tool.

Structured programming constrains *form* (syntactic tier). The semantic disciplines — SOLID, TDD, DDD — constrain *meaning for a human reader* (semantic tier). Generative Specification operates at the pragmatic tier: derivability by a *stateless* reader becomes the binding constraint.

2. The Abstraction Ladder

The ladder has been moving in one direction since the first compiler freed the engineer from machine code. Each step produced a more capable reader. Each more capable reader demanded a richer specification. The engineer stopped managing registers when compilers could derive machine code from expressions. Stopped wiring object graphs when frameworks could derive them from configuration. Stopped writing route handlers when annotations could declare them. The pattern — specify *what*, not *how*; let the reader derive *how* — has been the field's intuition for sixty years.

The reader changed again in 2017. A large language model reads context-sensitively: its interpretation of any token depends on everything surrounding it in the context window — the import statements, the class hierarchy, the architectural rules in the loaded documentation. These are not optional enrichments. They are the grammar the model uses to determine what a valid sentence in this system looks like. The specification can now operate at a higher abstraction level than the implementation language, stated in the language of the domain, derived into implementation by a reader that understands the surrounding context.

The recurring pattern the field has been executing — identify a layer where intent is still being prescribed as execution, find or build a reader capable of deriving execution from a richer specification, and remove the prescription — continues at the lifecycle layer. Architecture, decisions, conventions, rationale: the layer that human teams carried implicitly in shared memory. Generative Specification names the pattern, applies it to that layer, and derives the discipline that follows from the reader now available.

3. The Theoretical Gap: From Context-Free to Context-Sensitive Practice

The Introduction described why the reader changed. This section describes what that change costs when the specification does not change with it.

An AI coding assistant starts from the artifacts present. The context window resets at the session boundary, carries no institutional memory of prior sessions (Tulving, 1972; Squire, 1987), and has no mechanism for deterministic judgment — every output is a probability distribution over possible continuations. What the channel carries is itself a specification act: incoherent artifacts amplify incoherence at generation speed. Where a human engineer *interprets* an

underspecified requirement — compensating across the gap with memory, inference, and accumulated context — the AI processes what is present. The human compensation layer does not exist.

The missing context is not ambiguity in a present signal but absence of institutional memory. The gap that prosody and paralanguage bridge in human dialogue — "that doesn't feel right" carries a precise technical concern through everything the room provides — is a gap the specification must close explicitly. The cost of ambiguity is not misunderstanding but **drift** (analogous to architectural erosion; De Silva and Balasubramaniam, 2012): implementation that is locally valid and tests-passing but architecturally incoherent, propagated across every subsequent session that inherits the corrupted context. Lehman's laws (1980) establish that complexity increases unless active work is done to reduce it. Drift is that law operating in the absence of a specification constraint.

The specification does not merely describe the system. Processing it *causes* the system to be what it becomes. That is why GS must be designed for a **stateless reader**: an executor that begins each session with no memory of prior sessions, no institutional context, no accumulated conventions, and no ability to ask clarifying questions. Everything not in the artifacts is absent.

The expanding context window does not escape this conclusion. An infinite window over an underspecified codebase is not infinite derivability — it is an infinite drift surface. The model reads more of the implicit record; it cannot derive intent that was never externalized. The structural enforcement layer — commit hooks, CI gates, phase guards — is also independent of context size: a model with unlimited memory cannot prevent incorrect output from entering the codebase unless the discipline makes that output architecturally unreachable.

Concurrent infrastructure-layer work addresses an adjacent problem. Anthropic's Auto Memory feature for Claude Code (2026) writes session notes from agent corrections across sessions; Auto Dream, a background consolidation process, prunes stale or contradictory memories and indexes them after every five sessions. This is a bottom-up approach: it learns from observed behavior and consolidates. GS is a top-down approach: the architectural constitution is authored intentionally before generation begins. Both address the stateless reader problem at different layers — Auto Dream reduces preference drift across sessions; GS constrains architectural intent within and across sessions. They are complementary, not competing. The distinction matters: reactive consolidation learns what the practitioner does; preventive specification states what the system must be. A well-maintained GS artifact set is the stable substrate that makes memory consolidation coherent rather than arbitrary.

3.5 Related Work

Other researchers arrived at this problem independently, from different directions. It belongs here — after the problem has been stated, before the solution is defined — so readers can evaluate the distance between what others found and what this paper claims.

Two independent research threads validate the problem formulation, neither arriving at the paradigm claim or the full methodology.

Gordon (2024, ACM Onward!) argues in *The Linguistics of Programming* that linguistic research (including formal grammar theory and the Chomsky hierarchy) offers substantially underused conceptual tools for programming language and software engineering research. Gordon establishes structural parallels between linguistics and PL/SE without focusing on LLMs as the reader who changes the design requirements. This paper takes a specific step within the direction Gordon identifies: grounding the Chomsky hierarchy in the architectural shift produced by deploying LLMs as primary consumers of software specifications, and deriving the restriction discipline that follows. Gordon independently identifies the territory this paper's core framework inhabits, without arriving at the stateless-reader consequence, the restriction discipline, or the paradigm claim. The distinction is between analytical and prescriptive work: Gordon maps the domain; this paper derives a discipline from standing in it.

Thiolf (2025, KIT/KASTEL) independently identifies, in *Analysis of Project-Intrinsic Context for Automated Traceability Between Documentation and Code*, the exact failure mode described in §3: architectural drift caused by

implicit context that AI tools cannot access, observed empirically through documentation-code traceability gaps in AI-assisted development sessions. The problem statement matches without coordination. Thirolf proposes automated traceability tooling as a structural response — complementary to but narrower than the full generative specification methodology.

Orlanski et al. (2026) introduce **SlopCodeBench** (arXiv:2603.24755), a language-agnostic benchmark of 20 problems and 93 checkpoints in which agents repeatedly extend their own prior solutions. Findings: no agent solves any problem end-to-end across 11 evaluated models; structural erosion rises in 80% of trajectories, verbosity in 89.8%; agent code is 2.2× more verbose than matched human-authored code and deteriorates with each iteration while human code stays flat. A prompt-intervention study shows prompting improves initial quality but does not halt degradation. The authors conclude "current agents lack the design discipline iterative software development demands." This is independent empirical measurement of the failure mode §3 names, using trajectory-level instrumentation the GS experiments do not employ. The mechanistic explanation follows from tooling: each context-window compaction retains recently active files but discards prior architectural decisions — the agent continues from a lossy summary, applying the same token-pressure pattern from a degraded baseline. GS artifacts are structurally resistant because the architectural constitution and ADRs are short, structured, and loaded at session start; they fit the compaction routine's file budget. Session memory does not fit; it is what the summary replaces.

These works establish that the failure mode this paper addresses is not an artifact of a single practitioner's context. SlopCodeBench independently measures the problem GS is designed to solve but does not test GS as a solution and is not cited as solution validation.

Anthropic (2026) provides a fourth validation thread through Auto Memory and Auto Dream, shipped in Claude Code (March 2026). Auto Dream is a background consolidation sub-agent that prunes stale session notes, resolves contradictions, and reindexes the memory corpus between sessions — addressing the same failure mode §3 describes: accumulated session notes become noise that degrades output across multi-session workflows. This is a production feature shipped because the failure mode was observed at scale; it constitutes a category of evidence distinct from benchmark studies. The relationship to GS is architectural, not competitive: Auto Dream is reactive (it consolidates after drift has accumulated); GS is preventive (the architectural constitution and ADRs make drift structurally unreachable before any session begins). Both address the stateless reader problem at different points in the causal chain.

The convergence of four independent lines of evidence — two academic (Gordon, Thirolf), one benchmark (Orlanski et al.), one product engineering decision (Anthropic Auto Dream) — is precisely the finding §3's theoretical claim predicts.

The full treatment of GS's relationship to SOLID, clean architecture, TDD, DDD, the prior paradigm's anomalies, and the LLM code generation research literature is in §5.

Status of companion work (current as of April 2026)

Work	Status
DX2 — Transfer study	Design committed; execution May 8–12, 2026. Hypothesis: GS discipline transfers through artifacts alone, without facilitation or artifact-generation phase. Pre-registered before enrollment opens. Results to be integrated in §7.8.A.
Loom — Formal language layer	Active development (M1–M23 complete). Companion paper in preparation. All references in this paper route to the companion.
BIOISO — Biological Isomorphisms	Companion paper in preparation. Colony run in progress.

This paper stands on its own evidence. The companion works extend the claim; they are not required to evaluate it.

4. Generative Specification: The Principle

The preceding sections established why a new discipline is necessary: AI readers are stateless, specifications designed for human readers are insufficient, and the gap compounds at generation speed. This section defines what the discipline consists of.

Paradigm throughout this paper carries Robert C. Martin's precise sense: a discipline defined by what it *removes* from programmer freedom. Structured programming removed `goto`. Object-oriented programming removed unconstrained access to internal data. Functional programming removed variable reassignment. GS removes the freedom to leave architectural intent implicit. This removal differs in kind from its predecessors. Structured programming, OOP, and functional programming each constrained freedoms for human readers who could still compensate for gaps through memory, collaboration, and institutional context. GS's removal is absolute for its reader: a stateless executor that begins each session with none of those recovery mechanisms. What prior paradigms made inconvenient, GS makes structurally absent — because the reader that would have compensated does not exist. The semiotic tripartition — *syntactics, semantics, pragmatics* — follows Charles W. Morris (1938 *Foundations of the Theory of Signs*): the pragmatic tier is the relation of signs to their interpreters in context of use. GS occupies the pragmatic tier because it governs derivability for a reader who carries no interpretive context — the tier prior disciplines left vacant.

The most common alternative framing — that AI coding assistants already achieve what GS claims — conflates acceleration with direction change. Copilot, Cursor, and completion-based tools write code in the paradigm the practitioner already uses, faster. GS removes the obligation to write code at all. The remaining obligation — writing the grammar from which correct implementations are derived — is categorically different: different artifact set, different workflow, different failure mode. A faster completion tool is a Popperian refinement. GS is a Martin-sense paradigm: it removes a degree of programmer freedom in the same structural move structured programming removed `goto`. The practitioner who adopts GS does not speed up. They change direction.

Martin's three examples establish the mechanism; they are not its ceiling. Every formally proved theory of correct computing is a potential restriction layer: Hoare's precondition-postcondition logic (1969), Milner's type polymorphism (1978), Meyer's design by contract (1992), Girard's linear types (1987), Honda's session types (1998), Fielding's hypermedia constraints (2000). Each was abandoned not because its mathematics failed but because sustaining it required consistency no human team could maintain at scale. The AI executor holds all of them. The specification opens the door. The full argument for why these theories failed under human practice and succeed under GS is developed in *Onwards!*

On enforcement. A reviewer familiar with Martin's criterion — that paradigms are defined by the restrictions they *enforce*, not merely recommend — may object that GS's restriction is a document the compiler knows nothing about. The objection has a structural answer. GS's enforcement layer is not the specification file; it is the quality gate stack the specification mandates. The pre-commit hook that rejects a file exceeding the Bounded property's line limit is enforced — it blocks the commit. The CI pipeline that fails on a coverage drop is enforced — it blocks the merge. The MCP tool boundary that prevents a Bounded violation from being committed is enforced — it prevents the write. The spec is the grammar; the toolchain is the compiler. The architectural constitution is not advisory in a GS-governed project any more than a type annotation is advisory in TypeScript: both can be suppressed with deliberate override, and both are automatically checked without override. What GS restricts — implicit context, unbounded scope, unverified output, unrecorded decisions — is structurally unreachable in a correctly governed project, not merely discouraged. The enforcement is distributed across commit hooks, CI gates, and MCP boundaries rather than

concentrated in a single language parser, but enforcement is enforcement. What changes with each new model generation is not the paradigm's restriction claim; it is the precision with which the restriction can be stated. As models become more capable readers, some current directives become redundant and fall away. The restriction class itself — implicit context is not permitted in the lifecycle layer — does not change.

4.1 The Mechanism

A programming discipline of the pragmatic tier consists of making derivable from artifacts what was previously accessible only through interpretive context. That is the obligation a stateless reader makes structurally necessary.

4.1.a The Grammar Mechanism

A **Generative Specification** is a finite, coherent set of system artifacts sufficient to generate any valid implementation state of the system without requiring external human context.

A description records what was built. A specification is what a correct implementation must continuously fit. When the implementation drifts, you fix the specification and regenerate. The specification is the program.

The central structural property is *derivability*: a system's lifecycle layer is derivable when a stateless reader, given its artifact set alone, can correctly determine what should be built, where, why, and to what contracts, without requiring external human context.

Valid carries a broader obligation than Chomsky's *grammatical*: a valid implementation state is both structurally well-formed under the specification's rules and conformant to its behavioral and acceptance-test obligations. A wrongly-specified grammar is possible — correct by its own rules while failing the system's actual obligations. This is the methodology's primary failure mode, and the reason the specification faces the same verification discipline as the implementation it governs (§8.10).

Specification is the act of ruling things out. Without constraints, any output from the AI's vast default distribution is valid — which means every output is arbitrary. As constraints accumulate — naming conventions, architectural boundaries, ADRs closing open decisions — the set of valid sentences shrinks. But the AI's ability to derive *the correct sentence for a given requirement* grows. Restriction is the activation mechanism. The output-space formulation is precise: every constraint removes a degree of freedom from the space of valid programs. The programs that remain after all constraints are applied are exactly the correct ones — a smaller space where every reachable point is right. The restriction is the expansion mechanism.

The AI's training corpus contains the full formal tradition of computer science. Without specification, the model defaults to what human practice historically permitted: the convenient shortcut, the informal approximation, the discipline abandoned under deadline pressure. The practitioner who names Hoare contracts or session types is not teaching the AI anything new — they are opening the door to knowledge the model already holds. The depth of the specification determines the depth of the formal tradition activated.

Positional placement matters. Liu et al. (2023) demonstrate systematic accuracy degradation for information positioned in the middle of a long context window. The architectural constitution — placed at the leading position of every AI session — responds directly to this failure mode. The spec is the first thing the model reads.

The Chomsky hierarchy provides the structural analogy: as readers gain expressive power, the specification required to govern them correctly must become richer. Finite rules generating infinite valid outputs is the structural intuition. The practical implication is a single imperative: **assume nothing**. Every assumption is a gap the agent will fill arbitrarily, at generation speed, across every session that inherits the result.

4.1.b The Economic Consequence

The economic consequence of this structure is **cost inversion**: the cost of iteration approaches zero.

In a traditional development cycle, wrong output costs a sprint: read the code, identify the error, write a correction, review, wait for CI, merge. Coordination overhead accumulates at every boundary. Under GS, wrong output costs one sentence: identify the missing constraint, add it to the specification, re-run. The AI is stateless — it re-reads the complete specification and regenerates from scratch. No codebase navigation, no branch management, no review cycle for the specification change itself.

The residual is real: identifying *which* constraint is absent requires domain fluency. That cost is bounded by the practitioner's own competence, not organizational friction. The process is self-correcting: each iteration makes the grammar more precise, revealing the next gap.

At portfolio scale, cost inversion shifts the binding constraint from execution capacity to *specification bandwidth* (coined here: the rate at which intent can be correctly externalized into a durable specification). Multiple projects cycle concurrently — a project in a waiting state (deploy running, output under review) requires no execution from the practitioner. Portfolio size is bounded by status management, not execution load.

Once a grammar is complete, derivation is mechanical — not because the model is intelligent but because completeness closes the space of valid outputs to those that are correct. The adoption dynamic follows every prior software abstraction: friction on entry, then dominance once the productivity margin exceeds the residual cost. What differs is magnitude: the cost being settled is not a layer of implementation but the act of converting intent into implementation itself.

Formally correct specifications previously failed at scale — REST's hypermedia constraints, semantic web annotations, session types — because their annotation burden exceeded what human teams would sustain. The failure had six identifiable causes at two levels. *System-level*: (1) annotation fatigue — correct invariants existed but no team could maintain them under production pressure; (2) single-target economics — one annotation per language never paid for itself at the tool layer; (3) tooling fragmentation — type checkers, security auditors, dashboards, and configuration surfaces never unified into one reviewable artifact. *Practitioner-level*: (4) learning cost — no career was long enough to master the intersection of all relevant formal disciplines simultaneously; (5) maintenance erosion — disciplines known at hire degraded under deadline rotation and team turnover; (6) transfer loss — knowledge lived in people, not artifacts; when people left, so did the discipline. The AI executor eliminates all six simultaneously: it has no incentive to skip annotations (1), its training corpus covers every annotation for every target (2), it reads the unified discipline surface without fatigue (3–4), it applies every constraint consistently across sessions without erosion (5), and the specification artifact is the transfer mechanism, not the practitioner (6). Correct implementation becomes recursive: each correctly implemented standard makes the system more legible to the executor that built it, raising the quality floor for every subsequent generation.

The philosophical and civilizational consequences — why the formal tradition is now achievable at every scale, for every project — are developed in the companion essays *Onwards!* and *The New Golden Century*.

4.1.c The Convergent Principle

The grammar mechanism is specific to the relationship between three elements: a declared specification, an executor capable of producing outputs from it, and an observation mechanism capable of measuring the gap and triggering correction. Any system where all three elements are present operates under the same discipline: *the correctness of the outcome is a function of the completeness of the specification*. Its convergent form across other executor domains — industrial systems, autonomous vehicles, medical devices — is developed in §10 and *The New Golden Century*.

One consequence compounds forward: a correctly implemented GS system is more legible to the next stateless reader session that encounters it. The executor that generates correct output against a complete specification has, in doing so, made the system's formal properties visible in its own artifacts — correct naming, enforced boundaries, emitted decision records. Every subsequent session starts from a higher floor, because the reader of that output is the same kind of executor that produced it. Correct output is a recursive investment: it raises the quality of the next generation without any additional practitioner act.

4.1.d The Pedagogical Structure

GS does not instruct by exhaustion. The specification states what to build and why — intent and constraint, not procedure. The spec orients; the model derives. That division is the mechanism, not a limitation.

Correction arrives by exclusion. Quality gates name what is not acceptable and return the output — the cognitive apprenticeship model (Collins, Brown & Newman, 1989): the expert makes tacit knowledge visible through targeted correction, not prescription. Each gate is a named failure mode. As practitioners contribute gates derived from their own project failures, those failure modes propagate to every project that adopts the shared template. The floor rises (Dreyfus & Dreyfus, 1986: internalized failure modes become invisible rules; community-contributed gates encode that internalization structurally).

The validation strategy maps the same three-stage sequence. AX (Author-Executed): solo practitioner, unobserved. BX (Benchmark Cross-validation) and RX (Replication): calibration against independent implementations not shaped by the same assumptions. DX (Developer Experience): expert evaluation against criteria defined before the work was seen.

4.1.e Illustration: The Art Generation Pipeline

A strategy game concept given to the system as a narrative idea demonstrates the mechanism at a domain that has no prior GS tooling. The precision of the idea is the ceiling of everything that follows: a vague concept produces a generic game; a precise one — factions with named ideological conflicts, unit archetypes tied to faction doctrine, a color palette grounded in environmental lore — produces a specification the AI executes with fidelity. The AI derives toolchain selection, model configuration, quality constraints, and every generation step without human direction at any intermediate stage. The quality constraints *are* the specification: once they close the surface, every reachable output is a correct one.

A calibration phase is required before the pipeline reaches steady state: human review of sample outputs to determine whether the constraint set is sufficient. This phase can itself be automated — a vision-capable model given the stated quality rules and a sample can evaluate conformance and return a structured gap analysis. The human's final role is to decide the gap analysis is empty.

One surface this constraint vocabulary cannot close is aesthetic judgment: visual weight, compositional tension, emotional resonance. A constraint that closes symmetry and palette conformance produces a *technically correct* asset; whether it is compelling is a judgment the specification cannot make. The pipeline produces materials, not final deliverables — it collapses the distance between idea and revisable first form, which is precisely what an artist needs to begin. A failure mode specific to this structure — the AI building the sample artifact instead of the generative mechanism — is named and addressed in §8.15.

4.1.f The Seven-Tier Obligation Cascade

Generative Specification does not improve how code is written. Applied fully, it removes seven categories of work from the practitioner's responsibility entirely. Each tier 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. The formal constraint added to the system is precisely the mechanism by which the practitioner's obligation dissolves — the restriction is the liberation. Both statements are required to characterize a tier fully: what the system now must satisfy, and what the practitioner is therefore no longer required to provide.

The seven tiers form an obligation cascade — what GS removes from the practitioner, step by step:

Proven tiers (T1–T4): empirically demonstrated across production deployments and the DX1 practitioner study.

Tier	Obligation removed	Primary mechanism	Status
------	--------------------	-------------------	--------

T1	Write code	GS + ForgeCraft — spec is primary, code is derived	Proven (DX1, §7.8.A.1)
T2	Read or review generated code	Self-correcting harness — ForgeCraft gates, Hurl, mutation testing, automated scoring	Proven (DX1, §7.8.A.1)
T3	Manage infrastructure	Config/CLI-driven CI/CD; CAE/LTE/PRD environments; UAT, load, security, and gateway automation	Proven (Chronicle/Railway, §7.7)
T4	Monitor and maintain the living system	Chronicle signals + CodeSeeker → automatic anomaly detection and correction	Proven (COMPASS ETL, §7.7)

Future tiers (T5–T7): architecturally specified and in active development; empirical demonstration forthcoming.

Tier	Obligation removed	Primary mechanism	Status
T5	Expand and evolve the system	Bio Iso reads telos and signals; governed mutation, senescence, self-improvement	In progress (Loom colony, §7.8.A)
T6	Design the ecosystem	Axon derives a colony of T5 systems from a single problem statement — no architect required	Designed (Axon, §7.8.A)
T7	The process itself	Meta-telos observes the practitioner and determines what is needed before it is asked	Research agenda

T1–T4 are proven across production deployments. T5 is being demonstrated through the Loom colony simulation and multi-project Bio Iso deployments (results expected before arXiv submission). T6 is architecturally complete; empirical demonstration is the next validation phase. T7 is the logical terminus — stated as a research agenda rather than an implementation claim.

What is not a tier: Loom is a language layer that cuts across T1–T5. It is not a rung on the cascade; it is the medium through which the cascade eventually operates at the compiler layer. Full treatment is in the companion Loom paper.

Philosophical and civilizational frames (not tiers): Nous/Logos (§4.1.b) grounds why T1–T4 are structurally achievable now — the Logos that can hold the Nous without degrading it across sessions. The Golden Century (§8.8) names the civilizational consequence when T5–T7 complete. *Attention is All You Have* frames what remains uniquely human when T1–T7 are operational: the capacity to direct attention, not the burden of execution. Ambient Engineering (§9.2) names the interface mode at T6–T7 — the environment is the computer; intention accumulates without session cost. These frames are not tiers. They ground, contextualize, and project the cascade.

Tier 1 — The Industrial Blueprint: you do not write code. The specification constitutes the industrial blueprint — the source from which every implementation is derived and re-derived on demand. The testing disciplines (TDD, integration, E2E, mutation, contract, non-functional) each specify a distinct class of behavioral obligation: TDD per unit, integration at boundaries, E2E for telos fulfillment, mutation for detection completeness, contract for distributed agreements, NFRs as executable thresholds. A spec including only TDD has left every other class implicit — which is where the failures are. The executor holds every formal discipline in its training corpus without fatigue, erosion, or transfer cost; the structural files are the mechanism by which every new session starts from the same enforced position.

Tier 2 — Semantic Validation: you do not read the generated code. The behavioral contracts from T1 become the validation harness: API contract verification, Playwright use cases, headless simulations, log masks. Every executable

obligation is verified against the running system automatically. If validation fails, the specification is tightened and T1 regenerates.

Tier 3 — Environment and Lifecycle: you do not touch infrastructure. CI/CD pipelines, deployment manifests, environment configurations — governed by the same specification as the code. Compliance policy (HIPAA, PCI-DSS, GDPR, SOC2) is enforced as a deployment-gate type-level constraint, not a manual checklist reviewed at launch. The practitioner never issues a CLI command or edits an infrastructure file.

Tier 4 — Self-Monitoring and Self-Healing: you do not diagnose bugs. Logs, signals, and runtime observations are evaluated against the same formal properties that drove construction. Drift from the specification is a specification violation, detectable and correctable by the same derivation mechanism that built the system. Demonstrated in COMPASS/The Eye (§7.7).

Tier 5 — BIOISO: the system evolves. Self-maintaining, self-renewing formal systems whose lifecycle is governed by the specification. The theoretical framework is at bioiso.dev and in the companion paper *Biological Isomorphisms in Formal Self-Maintaining Systems* (in preparation). The T5 lifecycle detail — mutation triggers, reproduction protocols, telos as fitness grammar — is routed to that companion paper.

Tier 6 — Problem-Stated System Synthesis: you do not design the system. The practitioner states a problem. A stateless reader derives an interacting set of T5-governed programs — each with its own derived telos, interacting through typed channels, dying when their telos is fulfilled. The practitioner is no longer the system architect; they are the problem-holder. Formal prerequisites and colony simulation are specified in Loom's milestone roadmap and the *Bio Iso* companion paper.

Tier 7 — Meta-Telos: the process itself. A system that has observed the practitioner across their full history infers what is needed before it is asked. T7 is noted as the logical terminus — stated as a research agenda, not an implementation claim. Governance must precede capability; what T7 requires is not more formal theory but a careful answer to who decides what the practitioner needs before any autonomous inference becomes action.

On the role of the harness. The specification establishes intent; it does not certify that the derivation was faithful. That certification is the harness's function. A specification without a verification harness at T2 minimum is an assertion, not a guarantee: the claim "spec is the program" holds structurally only when the behavioral contracts from the specification are continuously verified against the running system. The harness is what closes the derivation loop — it is as constitutive of the GS guarantee as the specification itself. This is why T2 is not optional scaffolding but a structural requirement: removing it degrades the paradigm claim from a guarantee to a discipline preference.

The seven tiers are not independent. T2 validation requires T1 use cases be formal enough to test against. T3 compliance gates require T1 data flow labels. T6 synthesis requires T5 self-maintenance. Specification quality is the only constraint determining how far the cascade runs without the practitioner.

On cascade refinement. The tier hierarchy is conceptual; the implementation loop is recursive. A failure detected at T3 or T4 does not necessarily indicate a T3 or T4 defect: it frequently indicates a T1 gap — an NFR not stated, a data flow not labeled, a contract left implicit — whose downstream consequence became visible only at the higher tier. The correct response is to return to T1, close the gap, and re-run the cascade from that point. This non-linearity is not a weakness of the method; it is the expected diagnostic behavior of a system in which all higher tiers derive from the specification. T3 and T4 failures are detectors; T1 is almost always the site of correction. A complete NFR register at T1 is therefore not a documentation exercise but a prerequisite for T3 and T4 enforceability.

4.2 The Three-Tier Taxonomy

Throughout this section, syntactic and semantic are used in the programming-language sense: syntactic = pertaining to the form and structure of source artifacts; semantic = pertaining to the meaning those artifacts communicate to a reader who brings interpretive context. This is consistent with the Morris semiotic tripartition cited in §4 and with

standard usage in programming language theory, but differs from the technical senses these terms carry in formal linguistics.

The Syntactic and Semantic Tiers

Syntactic disciplines (Martin's three paradigms — structured programming, object-oriented programming, and functional programming, described above — and structural schema such as clean architecture, a layered design pattern that separates concerns into concentric rings: domain entities at the center, application logic surrounding them, infrastructure at the outermost edge) constrain the *form* of source artifacts: what constructs are permitted, what dependency directions are allowed. Whether every principle widely discussed as a paradigm falls neatly into this tier is a taxonomy debate this paper notes for completeness and takes no part in; the productive claim is directional: these disciplines constrain *what is permitted in the artifact*.

Semantic disciplines (SOLID — five principles for structuring object-oriented code: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion — test-driven development, domain-driven design, behavior-driven development, conventional commits) constrain the *meaning* that structure communicates to a human reader who brings context to the interpretation. A SOLID-violating codebase compiles; its cost is paid by engineers who recognize the deficit. TDD (Test-Driven Development — the discipline of writing a failing test before writing the code that satisfies it) makes a codebase *certifiable*: it removes the option of shipping unproven code, and the enforcement layer that makes it a discipline rather than a preference is real (CI — Continuous Integration — gates that automatically run tests on every commit, coverage requirements, deployment blocks). These disciplines assume a reader with state: colleagues, institutional memory, interpretive context built over shared history. TDD occupies the boundary between this tier and the pragmatic: a test suite is the closest prior art to a stateless, machine-readable behavioral contract, and TDD's verification posture is carried directly into GS as its Verifiable property (§4.3). What places TDD in the semantic tier is its incompleteness as a derivation grammar, tests certify behavior but leave architecture, naming, decision history, and rationale implicit. GS subsumes TDD rather than extending it.

The Pragmatic Tier

Generative Specification is a programming discipline of the pragmatic dimension, the first to name the obligation to make the lifecycle layer derivable for a context-sensitive stateless reader. It constrains not what is constructed and not what communicates to a reader with context, but what is *derivable by a reader with zero context*: no colleagues to ask, no institutional memory persisting across sessions, no informal channels through which intent can travel. Every intent that would previously have been resolved through shared knowledge must be externalized as formal artifact, because the channel through which shared knowledge travels does not exist for the stateless reader. The pragmatic tier had no prior occupant not because the distinction was unrecognized, and not because stateless readers did not exist, IDLs and formal specification languages are both stateless by design, and they predate LLMs by decades, but because no widely-deployed stateless reader *of the pragmatic kind* existed: one deployed to read lifecycle intent and derive from it what should be built, where, and why, without requiring a human to navigate the gap between the specification and the implementation. IDLs (Interface Definition Languages — specifications like OpenAPI that define what messages a system boundary accepts, without addressing how or why the system is built as it is) read interface contracts. Formal specification languages verify property invariants. Neither reads the lifecycle layer. The transformer architecture produced the first widely-deployed reader that does, and with its deployment, leaving the lifecycle layer implicit changed from a recoverable cost paid by skilled humans to a structural failure propagated at generation speed.

The lifecycle layer is the subject of the derivability obligation (coined here; the nearest established concept is specification completeness, as formalized by Parnas in 1972) GS states. It comprises: *architectural identity*, what the system is, how it is structured, and why; *evolutionary intent*, which directions of change are valid and which violate structural invariants; *quality contracts*, the behavioral, performance, security, and compliance obligations the system must satisfy; and *decision history*, what alternatives were considered and why they were rejected. The lifecycle layer excludes the type system, the test suite, and the source code, those belong to the syntactic and semantic tiers

respectively. A codebase that satisfies SOLID and has full test coverage is syntactically and semantically specified; it is not lifecycle-specified if an executor without institutional memory cannot determine from its artifacts alone whether a proposed change is architecturally valid.

On the Semiotic Origin of the Taxonomy

The syntactic/semantic/pragmatic trichotomy originates in Charles Morris's 1938 *Foundations of the Theory of Signs* — settled semiotic theory. Applying it to classify the obligation structure of programming discipline is this paper's proposal: a theoretical frame, not a claim within semiotic theory itself. The assertion that GS opens the pragmatic tier rests on the structural observation above — that no prior discipline stated the obligation to make the lifecycle layer derivable for an executor carrying no prior session context — not on the taxonomy alone. The classification provides orienting vocabulary; the taxonomy debate over which existing disciplines fall in which tier is noted and left open.

(Jim Gray's 2009 *The Fourth Paradigm: Data-Intensive Scientific Discovery* uses a paradigm count for scientific methodology — a distinct domain with no overlap in argument.)

Prior Occupants of the Pragmatic Surface

Interface definition languages (IDLs — OpenAPI, wire protocol schemas, RPC definitions) are stateless specifications designed for machine readers, but they operate at the interface layer: they define what crosses a boundary. A stateless IDL consumer can call an endpoint; it cannot determine whether that endpoint *should* exist or whether adding it violates an intentional boundary. Formal specification languages (TLA+, Alloy, Z notation) are stateless and system-level, but they are property verifiers, not derivation grammars: they establish that a design satisfies a stated invariant; they do not generate the naming convention, the module boundary, or the decision record the AI reads before implementing. More fundamentally, they were designed for a deterministic, rule-bound verifier — TLC, the Alloy Analyzer — that checks whether an explicitly modeled finite state system satisfies a stated logical property. That reader cannot reason about evolutionary intent or the direction of valid change, not because it is underpowered but because those questions are not expressible in the language it reads. The formal verifier and the context-sensitive natural language reader differ in kind. GS is designed for the second. No prior discipline was.

The Derivability Obligation

This discipline becomes visible when its failure mode becomes undeniable: The accumulated cost of leaving intent implicit in AI-assisted development (architectural drift produced at generation speed, propagating silently across every session that inherits a corrupted context) has been independently measured. Orlanski et al. (2026) find that structural erosion rises in 80% of AI agent trajectories and that prompt-intervention improves initial quality but does not halt degradation — "current agents lack the design discipline iterative software development demands" (§3.5).

The concept has roots in classical software engineering. Parnas (1972) established that a well-decomposed system should make every design decision locatable by inspection: a reader with access to the specification should be able to derive the intended behavior without consulting the implementation. Jackson (2001) extended this with the Problem Frames approach: the specification must be sufficient to bound the problem, or the implementation will fill the gap arbitrarily. The derivability obligation is the GS instantiation of both principles, applied to the AI generation context.

[^4]: The architectural constitution is agent-agnostic as a concept; agent-specific filenames are enumerated in the artifact grammar table (§6). The case studies in this paper use `CLAUDE.md` because Claude was the primary agent throughout. The paradigm's claim holds regardless of which agent or filename is used; these are interchangeable implementations of the same production rule.

The pragmatic tier has a specific failure mode that names it. A system that leaves context implicit is not merely poorly documented: it is *underspecifiable*: an agent with no persistent context cannot derive correct output because the grammar is incomplete. The consequence (architectural drift at generation speed) is structural, not stylistic. The distinction between the semantic and pragmatic tiers is therefore not one of intensity but of kind: semantic disciplines produce worse systems when violated; a pragmatic violation produces a grammar the context-free executor cannot parse, the failure is not a quality deficit at higher intensity but a derivability collapse.

A system has achieved generative specification when its artifact set is designed so that any AI coding assistant, given access to those artifacts alone, has what it needs to: correctly identify what should and should not change for any given requirement; produce output that conforms to the system's architectural, quality, and behavioral contracts; and detect when any existing artifact violates those contracts.

Whether a given AI model succeeds in practice is an empirical question about the model. Whether a given artifact set satisfies this design criterion is a structural question about the specification, answerable by inspection against the seven specification properties below.

Generative specification is a stronger property than "well-documented code." Documentation can be narrative and passive: it can exist in a README that three people have read and that the AI session will never be given. Generative specification is active: the artifacts are themselves executable, verifiable, and self-correcting. The distinction is operational: a system cannot violate a generative specification without a mechanism triggering.

4.3 The Seven Specification Properties

The seven properties below are to Generative Specification what SOLID is to object-oriented programming: a named, teachable set of obligations that makes the discipline concrete, inspectable, and transferable. SOLID tells a developer how to structure objects for a human reader who brings context and judgment. These seven properties tell a practitioner how to structure specifications for a stateless reader who brings neither. Each property names a specific failure mode observed in production across six projects — not a taxonomy constructed in advance, but a record of what breaks and why. Together they operationalize the derivability obligation (§4.2): make each class of failure structurally unreachable. The rubric derived from them — 0/1/2 per property, 14 points total — is the primary measurement instrument of the validation experiments in §7.

****Self-describing.****The system explains its own architecture, decisions, and conventions from its own artifacts. No external knowledge is required. Self-describing addresses the *rationale layer*: not just what the structure is, but why it is that way and what rules govern it. It extends the Single Responsibility Principle (Martin, 2002) from runtime modules to the full artifact surface: specification, tests, and architecture documents are each responsible for one concern and contain what a reader carrying no prior session history needs to understand that concern. Automatable checks: (1) presence of an explicit intent statement; (2) presence of a scope boundary statement. A specification lacking either fails regardless of prose quality.

****Bounded.****Every unit of work has explicit scope and seams. Functions do one thing. Modules own one concern. The line limit carries a mechanical justification: AI tool read operations are capped at a fixed line budget — a file that exceeds it is silently truncated, and the agent edits against an incomplete view. A specification artifact that exceeds the tool's read budget is, from the executor's perspective, equivalent to one that does not exist. Anthropic's MEMORY.md index is hard-capped at 200 lines for exactly the same reason: the startup context cannot load what exceeds it. The 300-line specification limit and the 200-line memory index cutoff are the same constraint at two layers. Bounded addresses the *structural layer* — the distinction from Self-describing: a well-annotated system with blurry module boundaries fails Bounded; a structured system with no architectural constitution fails Self-describing. The property operationalizes Parnas's information hiding principle (1972) at the specification level.

Bounded: The Sentinel Navigational Tree. At scale, Bounded applies to the session context itself. Loading all specification artifacts into a single context window produces the same pathology the property prevents at the file level — and, per Liu et al. (2023), degrades accuracy for information not near the leading position while consuming token budget on context the current task does not need. The structural solution is a **sentinel navigational tree**: a hierarchy of specification files where each node declares its own scope and routes to children. The root is always loaded (must stay within the bounded line limit); the AI descends only the path relevant to the current task. The tree is lossless — joining all leaf nodes yields the full specification — but each session receives only the slice it needs, eliminating both degradation and unnecessary token cost. Every well-formed tree must collectively contain five categories:

Category	What it covers
Architectural identity	What the system is, scope boundary, ADR index
Standards	Naming, commit discipline, quality gate thresholds
Constraints and prohibitions	What must not happen; boundary violations the AI must refuse
Tool sequencing	When to use which tool, in what order — not "these tools exist" but "use X before Y when C"
Routing	What each child covers and when to descend

Tool sequencing is the most commonly absent and most consequential gap. A spec that lists tools without stating when to prefer one over another forces unreliable inference.

Verifiable. The correctness of any output can be checked without human judgment. Types, tests, lint rules, coverage gates, and schema contracts form a continuous verification layer. The structural necessity of this property is visible at the tooling layer: AI coding agents report file-write success when bytes reach disk, not when the resulting code compiles. The success signal is write-completion, not semantic validity. Without an explicit verification layer, the agent's "done" and the developer's "done" refer to different states. Verifiable closes this gap structurally: the verification layer is not optional post-work, it is the definition of completion. Verification is automatic, fast, and blocking; not aspirational. In a GS context, the test suite carries an adversarial role: tests are written against interfaces, not implementations — to detect violations of the contract, not to confirm the current implementation. A test that verifies internal state fails on correct refactors and passes on behavioral violations that preserve internal structure. Load tests, penetration tests, and chaos probes are specifications of adversarial conditions with explicit acceptance thresholds (§8.11). Verifiable establishes that the check infrastructure exists; whether the implementation passes those checks in a real execution environment is the Executable property, scored separately.

Defended. Destructive operations are structurally prevented rather than merely discouraged. Commit hooks, branch protection rules, format enforcement, and MCP tool boundaries make certain classes of mistake architecturally unreachable. The system rejects malformed input the way a parser rejects a syntax error. The property formalizes what is informally called defensive programming and CI/CD hardening (Forsgren et al., 2018 DORA metrics) into a specification obligation: gates are not optional CI ceremonies; they are structural constraints on what the system may become.

Continuous integration pipelines can verify six of the seven specification properties automatically. Defended is the exception: whether adversarial challenge has been anticipated and answered requires human review. This is a hard ceiling on automated compliance checking, and §7 results should be read with this constraint in mind.

Defended: Process. This structural logic extends to the development process itself. Test-driven development requires a strict phase sequence: failing test, confirmed failure, then implementation. In a human workflow, temporal separation enforces this gate. A generative agent in a single context window has no such separation, the agent that will write the implementation is already present when it writes the test. This is not a discipline failure; it is a structural one: *phase-collapse* (coined here: the structural phenomenon in which the RED phase of TDD ceases to exist when test authorship and implementation authorship occur in the same context window), the RED phase ceasing to exist as a distinct moment because no temporal barrier separates test authorship from implementation authorship. An agent told to "write a failing test first" can comply in grammar while violating the property substantively, it will write a test shaped to fail against a not-yet-existing function, then immediately create that function. Instructions cannot close this gap. Only structural gates can. Forbidden patterns in the architectural constitution can prohibit implementation choices before a failing test commit is certified. A TDD workflow skill can require pasted test output as a mandatory

stop gate before phase advance is permitted. A pre-commit hook can reject a test-only commit where all tests pass, the signature of post-hoc or vacuous tests added after the fact. The [RED] commit naming convention makes TDD phase sequence machine-readable in the git log: a CI rule can detect a feat: commit without a preceding test: [RED] commit and block the merge automatically. Applied to process rather than artifact, the Defended property means the RED phase cannot be bypassed any more than a malformed commit message can be pushed.

Defended: Consequence Classification. In zero-tolerance execution domains (surgical systems, autonomous vehicles, signed legal instruments), Defended acquires a second obligation: the specification must classify the *consequence tier* of each executor action — which operations are reversible, recoverable, or irreversible — and name the human confirmation gate required before any irreversible action may proceed. In software (\$C_i \approx 0\$, \$R \approx 1\$) iteration absorbs residual errors; the classification is unnecessary. Where the correction loop cannot run after the fact, "do no harm" is a specification obligation. The deployment gate framework in §9.4 formalizes when an executor may be trusted with consequential action.

Auditable. The current state of the system, and the history of how it arrived there, is fully recoverable from the artifacts alone. Conventional atomic commits form a typed corpus of change. Architecture Decision Records document why the grammar evolved. Status files record the current implementation state. Nothing requires asking someone who was present at the time. Without an auditable trail, the AI will treat intentional architectural tradeoffs as defects to correct, producing drift silently, across every session that inherits the corrupted context. Full recoverability requires that commit discipline and the ADR record are both maintained. A specification without commit discipline provides partial auditability, behavioral contracts survive, but the reasoning behind session-level decisions does not. The Shattered Stars case (§7.6) demonstrates this boundary precisely: the spec held the system's structural contracts across sessions; what it could not hold was the provenance of the decisions that shaped them. The property elevates Architecture Decision Records (Nygard, 2011) and conventional commit conventions from recommended practice to required production rule: a system that cannot be audited for why it is the way it is has not satisfied the specification.

Composable. Units can be combined and extended without unexpected coupling. Clean architecture's dependency inversion and the pure function model from functional programming ensure that composition is predictable. The AI can work on any unit without unexpected propagation effects because isolation is structural, not assumed. (Complete independent isolation of any unit additionally requires the Bounded property: predictably-scoped context ensures that isolation holds across seams, not merely within them.) The property applies Clean Architecture's Dependency Inversion Principle (Martin, 2017) to the generation context: components must be navigable in isolation, so that a stateless reader can locate the relevant boundary without traversing the full artifact set.

The structural isolation Composable requires also determines the safety of cross-cutting changes. Text-pattern search tools available to AI agents are not AST-aware: they match strings, not symbols. A function rename or interface change propagates to callers, re-exports, barrel files, and dynamic imports — none reliably located by grep. In a system with clean Bounded module boundaries, the change surface for any interface modification is exactly the boundary declaration. The Bounded and Composable properties together close the search problem that AST-less tooling leaves open — the structural justification for graph-aware, call-chain-traversal tooling as a first-class GS component.

Executable. The generated output satisfies the behavioral contracts the specification defines when exercised against a real execution environment — not merely compiles and passes static analysis. Verifiable establishes that correctness checks exist; Executable establishes that the implementation actually passes them against a live execution context. A system can be fully Verifiable — correct types, passing lint, well-structured tests — while producing a server that fails every integration test against a real database.

Executable is scored conditional on specification availability: a formal contract (Hurl suite, OpenAPI diff, HL7 FHIR runner) enables automated measurement; a goal-directed program requires human acceptance criteria and is scored N/A. The adversarial experiment series (§7.8.B) articulated Executable as a distinct property by measuring the gap: treatment-v2 achieved 12/12 on six structural properties while only 1/9 test suites passed materialization. The

property formalizes what practitioners were already doing — verify-and-correct loops — so it can be specified, gated, and tracked.

The seven specification properties are universal, they apply to every project regardless of type or domain. Their concrete artifact expression, however, is project-type-parameterized: the specific quality gates, constraint vocabulary, and required artifact types that satisfy each property vary by what the project is. A healthcare system satisfying Defended requires PII redaction rules and audit logging constraints that a CLI tool does not; a real-time system satisfying Bounded requires latency contracts that a batch pipeline does not. §6 develops the artifact grammar and describes how the universal base and project-type overlays compose.

4.4 Contract Sufficiency: The What-How Distinction

Generative Specification governs *what* the system must do — behavioral contracts, acceptance obligations, architectural boundaries, non-functional requirements — and *why* the non-obvious decisions were made. It does not govern *how* those obligations are fulfilled. Multiple valid implementations can satisfy the same contract. A function that retrieves a user by email may use an indexed SQL query, a cache lookup, or a key-value store — each conforming to the behavioral contract, each differing in its mechanism.

Non-functional requirements belong unambiguously to the *what* layer. A latency threshold, memory budget, throughput floor, or security classification is an obligation, not an implementation choice. At runtime, each NFR becomes a quantified acceptance criterion: a load test asserting p99 latency under 200ms, a penetration test asserting no known vulnerability class passes. The only failure mode for NFRs under GS is omission — an NFR not stated cannot be enforced.

This is the principle TDD operationalizes at the function level. A test specifies what a function must do; dozens of implementations can satisfy it. GS applies the same separation at the system level: the specification certifies what a valid implementation state is; the AI generates the *how* within that certified boundary. The correctness criterion is convergence, not inspection.

This will produce defects. The defect modes of GS differ structurally from those of traditional development: they arise from specification incompleteness rather than the coordination failures that dominate current practice. Specification error is auditable, correctable, and does not compound silently across team rotations.

The ratchet does not reverse. Every defect resolved produces a test and a permanent production rule; every ambiguity resolved becomes an ADR that closes an open decision forever. The defect is not evidence the method failed — it is a specification query: *what constraint, had it been present, would have ruled this out?* When that constraint is written, the grammar expands, and the class of output that produced the defect becomes unreachable.

What GS Does Not Govern: Prompt Engineering

A complete specification makes prompt engineering unnecessary. If the spec is complete, the stateless reader derives the correct output from the grammar alone. If few-shot examples are needed, the specification is incomplete: the example compensates for a constraint not yet stated. The correct response is to write that constraint. Prompt engineering techniques are also model-specific and version-specific; a specification is model-agnostic. The practitioner who needs examples is receiving a diagnostic: the spec has a gap.

5. Related Work: Relationship to Existing Principles

Generative Specification does not replace the syntactic and semantic tier disciplines. It operates at the pragmatic tier: it constrains derivability for a stateless reader, a requirement the prior disciplines were not designed for because no widely-deployed stateless reader existed at their formulation. Each prior discipline removed a programmer freedom and in doing so made the code more predictable for its intended reader. The table below maps each discipline to the tier it occupies, the freedom it removes, and the GS property it satisfies — or the gap it leaves that GS fills.

Discipline	Tier	What it is	Freedom removed	GS property satisfied	What GS adds
Structured programming	Syntactic	Prohibition on unstructured jumps (goto), replaced by loops and conditionals	Unstructured control flow	—	Pragmatic layer obligation: the reader that executes the work must be able to derive intent from artifacts alone
Object-oriented programming (OOP)	Syntactic	Encapsulation of data and behavior inside objects with explicit boundaries	Direct access to internal data structures	Bounded (partially)	Self-describing: the boundary must be externalized in artifacts visible to a stateless reader, not just respected in code
Functional programming (FP)	Syntactic	Prohibition on mutable shared state; computation as transformation of values	Variable reassignment across shared state	—	—
Clean architecture	Syntactic	Layered design pattern separating concerns into concentric rings: domain entities at the center, application logic surrounding them, infrastructure at the outermost edge	Arbitrary dependency direction between layers	Bounded, Composable	Self-describing: why the layers exist and which directions of change are valid must be in the specification, not deducible from the shape alone
SOLID	Semantic	Five object-oriented design principles: Single Responsibility (one reason to change), Open/Closed (open for extension, closed for	Tangled responsibilities and tight coupling between modules	Bounded, Composable	Self-describing: a responsibility boundary identified by SOLID says nothing about whether that

		<p>modification),</p> <p>Liskov Substitution (subtypes behave as their supertypes),</p> <p>Interface Segregation (no client forced to depend on methods it does not use),</p> <p>Dependency Inversion (depend on abstractions, not concretions)</p>			<p>boundary must be externalized in artifacts a stateless reader can inspect. A generative specification requires the system to describe itself completely, because the reader cannot ask a colleague</p>
TDD — Test-Driven Development	Semantic	<p>Discipline of writing a failing test <i>before</i> writing the code that satisfies it; CI gates enforce it by blocking merges on test failure</p>	Shipping code whose behavior is unverified	Verifiable	<p>Executable: a passing test suite certifies behavioral contracts against static assertions; GS adds runtime measurement against a real execution environment. Tests certify intent; Executable certifies derivation</p>
DDD — Domain-Driven Design	Semantic	<p>Methodology for aligning the software model with the business domain through shared language (ubiquitous language), explicit boundaries (bounded contexts), and domain-first modeling</p>	Domain incoherence and leaky abstractions across module boundaries	Bounded	<p>Auditable: DDD defines what the domain model is; GS additionally requires that the decisions behind it — what alternatives were considered and why they were rejected</p>

					— live in the decision record, not only in the code
BDD — Behavior-Driven Development	Semantic	Specification of system behavior in structured natural language (Given/When/Then) readable by both technical and non-technical stakeholders	Unspecified user-facing behavior: code that passes tests but satisfies no stated user intent	Verifiable	—
Conventional commits	Semantic	Typed commit message format (feat, fix, chore, refactor, etc.) that encodes the nature and scope of every change in the git log	Untyped, unreadable change history	Auditable	Elevated from recommended practice to required production rule: an unrecorded architectural decision is a gap in the grammar, not a style choice
CI/CD gates — Continuous Integration / Continuous Deployment	Enforcement	Automated pipelines that run tests, linters, coverage checks, and quality gates on every commit, blocking merges that fail	Post-hoc quality verification after code ships	Defended	Classified consequence tiers in zero-tolerance execution domains: the specification must identify which executor actions are irreversible and require human confirmation before proceeding

The practical implication: a codebase that follows SOLID and clean architecture is a necessary but not sufficient generative specification. It becomes sufficient when the self-describing, auditable, and executable artifact layers are present and maintained.

GS's four new contributions — the properties no prior discipline named — are worth stating plainly. **Self-describing**: the system must externalize its complete architectural identity in artifacts a stateless reader can inspect without asking a colleague. **Defended**: the process must have structural guards that make certain failures unreachable, not merely discouraged. **Auditable**: every architectural decision must be recorded as a required production rule, not a recommended practice. **Executable**: the implementation must be measured against a real runtime environment, not only against static test assertions. Together these four close the derivability gap the prior disciplines leave open.

GS is categorically distinct from prompt engineering — HOW-layer techniques that guide model behavior toward a particular implementation path. The full distinction is in §4.4.

Industry Prior Art: Spec-Driven Development

The practitioner community independently converged on the same starting insight: specifications should drive AI-assisted development, not follow it. **Speckit** (GitHub, 2025) and **OpenSpec** (@fission-ai/openspec , 2025) both implement this as a structured path from "write a spec, then prompt." **AWS Kiro** (2025) and **Tessl** (2025) implement spec-first development as the primary authoring surface, extending the same convergence to cloud-native and low-code tooling. **Fowler (2026)** surveys the emerging tooling landscape, establishing that the practice has reached mainstream visibility: when the field's leading methodologist publishes comparisons, the practice is past the early-adopter phase. A peer-reviewed taxonomy has appeared: **arXiv:2602.00180** (Feb 2026) defines three Spec-Driven Development rigor levels — spec-first, spec-anchored, and spec-as-source — the first formal classification of the practice this paper claims to theorize. **ThoughtWorks Technology Radar (2025)** placed SDD in the Adopt ring, its strongest practitioner endorsement, confirming independent spread across the field.

The divergence from GS is depth. Every listed system stops at Tier 1 (spec drives implementation). None addresses the verification harness (T2), infrastructure governance (T3), self-monitoring (T4), or evolutionary lifecycle (T5+). None formalizes the structural properties an artifact set must satisfy to make derivation correct. The shared starting point validates the problem is real; the divergence in depth is where the paradigm claim lives. The industry is converging on the practice without having named the principle. GS names the principle.

Industry Prior Art: Context Enhancement

Agent instruction files (`CLAUDE.md` , `AGENTS.md` , `.cursorrules`) and memory/status files are the field's first-order responses: inject rules into the context window, persist state across sessions. GS makes full use of them. But they are context-enhancement tools — they improve what enters the channel. None specify what the channel must be able to *derive* from what it receives. Feeding a richer window into an underspecified system produces richer drift at generation speed. GS is categorically distinct: not what to put in the channel, but what a complete grammar for a stateless reader must look like.

Independent corroborating work is covered in §3.5.

The Prior Paradigm and Its Anomalies

Agile iteration, TDD, CI/CD, and DDD constitute a mature layered methodology the industry converged on over two decades. Each discipline removes a degree of programmer freedom; each removal makes the system more predictable. Together they set the baseline GS extends.

The methodology breaks at generation speed. Each discipline assumes a human author: a developer who carries architectural intent between sessions, who notices when a new module violates an existing boundary, who understands why a decision was made three months ago. The quality checks are periodic; the generation is continuous. Drift accumulates in the interval between checks, locally invisible, propagating silently.

The anomalies are specific: TDD's RED phase ceases to exist when test and implementation authorship occur in the same context window (*phase-collapse*). Code review becomes structurally unable to catch drift that accumulated across ten sessions. ADRs become orphaned — the AI has no access to why the system is the way it is. Peng et al.

(2023) document a 55.8% productivity gain from AI assistance; the same generation capacity accelerates drift. The methodology that governed human-speed development has no mechanism for generation-speed incoherence.

GS extends the prior paradigm to the generation context. The prior paradigm removed degrees of programmer freedom at human speed. GS removes degrees of generator freedom at generation speed. The gap was not visible when developers wrote every line. It became visible the moment they stopped.

LLM Code Generation Research: Empirical Grounding

HumanEval (Chen et al., 2021) measures single-function synthesis in isolation; **SWE-bench** (Jimenez et al., 2024) measures AI patching ability on navigable codebases. GS addresses a structurally different problem at a prior layer: the conditions under which a codebase is navigable by a stateless reader in the first place. A model scoring 90% on HumanEval can still produce architectural drift across ten sessions if no specification governs cross-session structure. Whether GS-compliant codebases achieve measurably higher SWE-bench patch success is a direct empirical falsification candidate.

Peng et al. (2023) establish a 55.8% task-completion speed increase under AI assistance. GS's claim is not that AI improves speed — that is established. The claim is that speed without structural governance produces drift faster, and GS is the governance layer that makes multi-session speed sustainable.

Prompt engineering (White et al., 2023) establishes that input framing significantly affects output quality. A prompt is a session artifact; it governs one interaction. GS is the layer that persists across every session and governs every prompt submitted against it. The full distinction is in §8.9.

The seven GS properties have structural analogs in **ISO/IEC 25010** (Maintainability, Testability, Analyzability) — independent convergence from a standards direction, establishing that the properties are not ad hoc.

A separate empirical thread provides independent evidence for the paper's central mechanism claim — that the annotation burden blocking the formal tradition for decades is dissolving. **Stanford Clover (2024)** demonstrates closed-loop verifiable code generation: an LLM generates code and formal annotations simultaneously, a verifier checks them, and the loop iterates until verification passes — achieving an 87% acceptance rate on standard benchmarks. **Microsoft DafnyBench (2024)** demonstrates LLMs auto-annotating Dafny programs with formal invariants: 68% baseline accuracy rising to 98% with verifier feedback. **PropertyGPT (Ye et al., NDSS 2024)** generates formal verification properties for smart contracts via retrieval-augmented LLMs, discovering previously unknown vulnerabilities in the process. None of these systems is GS-aware. None frames itself as a paradigm shift. That is precisely their value: they are independent empirical proof that the annotation burden which made the formal tradition impractical for human teams is structurally gone. The formal disciplines did not fail because their theory was wrong. They failed because maintenance exceeded human capacity. That capacity constraint has changed.

6. The Artifact Grammar

The methodology's empirical record does not depend on ForgeCraft: any practitioner producing the same artifacts by hand produces the same grammar the AI reads. See §7.8 for authorship and conflict-of-interest disclosure.

A system built to generative specification consists of the following artifact types, each functioning as a distinct production rule in the system's grammar.

Artifact	Linguistic Analog	Function in the System
Architectural constitution ^[^4]	Grammar rules	Defines what is and is not a valid sentence in this system. Every AI interaction is governed by this document. Agent-specific filenames: CLAUDE.md (Anthropic Claude), AGENTS.md (OpenAI), .cursorrules /

		<code>.cursor/rules/</code> (Cursor), <code>.github/copilot-instructions.md</code> (GitHub Copilot), <code>.windsurfrules</code> (Windsurf). The concept is agent-agnostic; the filename is not. In projects large enough to exceed a single bounded file, the architectural constitution is the root node of a sentinel navigational tree (§4.3 Bounded).
Sentinel navigational tree	Grammar index with lazy derivation	A hierarchy of scoped specification files. The root is always loaded; each child node declares its own domain and routing condition; the AI descends only the path relevant to the current task. Joining all leaf nodes yields the complete specification. Five categories must be collectively present across the tree: architectural identity, standards, constraints and prohibitions, tool sequencing (when to use which tool under which condition), and routing. The root node must stay within the bounded line limit precisely because it is always loaded. A tree that omits any category from its leaves forces the AI to infer that category — which is a derivability failure at the navigation layer.
Architecture Decision Records (ADRs)	Etymology and rule changelog	Documents why the grammar evolved. Prevents the AI from "correcting" intentional decisions that appear suboptimal without context.
C4 diagrams / structural diagrams (PlantUML, Mermaid)	Syntax tree	The parsed structural representation of the system. Context at a glance for any agent entering the codebase.
Use cases, flow diagrams, sequence diagrams, state machine diagrams	Sentence patterns and grammar rules with temporal order	Each diagram type constrains a distinct dimension: sequence diagrams fix the protocol between components (which calls, in which order, with which contracts); user flow diagrams define the expected journey from entry point to outcome (and are simultaneously the script for every E2E test in that flow); state machine diagrams enumerate valid states and transitions (and directly generate state transition test cases and the user-facing documentation of each mode). These are not illustrations. They are production rules the AI reads before generating any artifact the diagram describes.
Schema definitions (database, API, event)	Type system / lexicon	The vocabulary of the system with its constraints formally stated.
Living documentation (derived)	Compiled output from the grammar	Documentation regenerated from the specification: OpenAPI/Swagger from type annotations or route decorators, TypeDoc/JSDoc from inline documentation, Storybook from component specifications, generated README sections from centralized specs. Documentation maintained separately from the code it describes is a liability: it will drift. Documentation derived from the same artifacts the AI reads is always current, because it shares a source of truth with the implementation.

Intentional naming conventions	Word choice	Semantic signal at every token. A function named <code>calculateMonthlyCostPerMember</code> carries domain, operation, unit, and scope. <code>processData</code> carries nothing.
Package and module hierarchy	Phrase structure rules	Communicates responsibility and ownership through structure. The location of a file is a claim about what it is.
Conventional atomic commits	Typed corpus with morphology	<code>feat(billing): add prorated invoice calculation</code> has a part of speech, a scope, and a semantic payload. The git log is a readable history of how the grammar evolved and why.
Test suite (TDD / adversarial)	Semantic validation + adversarial probe	Each test is a statement about what the system must do: a specification assertion <i>and</i> an adversarial challenge, the agent writes tests intended to expose incorrect code, not to document assumptions. The full suite is a continuously-running audit and a standing challenge to the implementation.
Commit hooks and quality gates	Parser rejection rules	Malformed input is structurally rejected before it enters the system. The architecture makes certain mistakes unreachable.
MCP tools and environment tooling	Runtime environment	The tools available to the agent define what operations are possible. Bounded tool access is bounded agency.

The artifact grammar above is the universal base; a complete generative specification composes it with a project-type overlay (healthcare adds PII rules and audit trails; real-time adds latency contracts; a game adds asset quality gates). ForgeCraft-MCP implements this through its tag system: every project receives `UNIVERSAL`, and each active tag (`API`, `WEB-REACT`, `GAME`, `FINTECH`, etc.) applies its overlay. ForgeCraft is **transitional scaffolding** — it enforces at the governance layer what Loom will eventually enforce at the language and compiler layers; its ADRs become verified contracts, its commit hooks become type-checker passes. ForgeCraft bundles CodeSeeker as a first-class component because the grep-versus-AST limitation (§4.3 Composable) makes agent-driven refactoring unsafe without graph-level navigation; governance and navigation are designed to operate together.

6.1 A Generative Specification in Practice

The artifact types above are not theoretical constructs. The following is a representative excerpt from the architectural constitution (`CLAUDE.md` in the Claude convention^[4]) that governed the SafetyCorePro refactor described in §7.1, written in its entirety before a single implementation change was made. The complete document is 155 lines.

```
# CLAUDE.md. SafetyCorePro

## Project Identity
- Primary Language: TypeScript 5.x
- Framework: Next.js 14 (App Router) + Prisma 5 + PostgreSQL
- Domain: Occupational Safety Management Platform
- Sensitive Data: YES: PII (employee records), safety incident data, compliance records

## Architecture Rules
- All data access goes through service/repository layers, never direct Prisma calls from components or route handlers.
- No business logic in API route handlers, they delegate to services.
- Multi-tenant: Every query MUST include `cuentaId` filter.
```


Never expose cross-tenant data.

- Permission checks via `requirePermission()` / `requireAuth()` as first line of every server action.

Layered Architecture

Pages / API Routes / Actions	← Thin. Validation + delegation only.
Services (Business Logic)	← Orchestration. Depends on interfaces only.
Domain Models / Types	← Pure data + behavior. No I/O. No framework.
Repositories / Adapters	← All external I/O (DB, APIs, files, queues)

Never skip layers. Dependencies point downward only.

Error Handling

- Custom error hierarchy per module. No bare `Error` throws.
- Errors carry context: IDs, timestamps, operation names.
- Fail fast, fail loud. No silent swallowing of exceptions.

Code Standards

- Maximum function length: 50 lines. Maximum file length: 300 lines.
- Every public function must have JSDoc with typed params and returns.
- No abbreviations except universally understood (`id`, `url`, `http`, `db`, `api`).
- Bilingual naming: Domain entities keep Spanish names (`visita`, `empresa`, `hallazgo`, `reporte`) to match DB schema. All technical code uses English.

Testing Pyramid

- Overall minimum: 80% line coverage
- New/changed code: 90% minimum
- Critical paths: 95%+ (permissions, multi-tenant isolation)
- Every test name is a specification: `test_rejects_duplicate_empresa`, `not test_validation`

Commit Protocol

- Conventional commits: `feat|fix|refactor|docs|test|chore(scope): description`
- Commits must pass: TypeScript compilation, lint, tests.
- Keep commits atomic, one logical change per commit.
- Update `Status.md` at the end of every session.

The AI read the architecture rules and produced services. It read the error handling rules and produced a custom exception hierarchy. It read the bilingual naming convention and applied it consistently across every new file. The specification is not a description of what was built. It is the grammar from which the build was derived.

6.2 The Initialization Cascade

The artifact types above are not produced in parallel or in arbitrary order. Each artifact is both an output of what precedes it and a production rule for what follows. A sequence diagram that contradicts the architecture is evidence that one of them is incomplete; an ADR written before the architecture is speculation rather than decision record. The initialization cascade is:

1. **Functional specification**, user-facing behavior, domain model, key entities, and system boundaries stated with enough precision that an agent can distinguish an in-scope request from an out-of-scope one. This is the axiom set; everything else is derived from it. If a requirement cannot be stated here, it is not yet a requirement.
2. **Architecture document**, the layered structure, module boundaries, and integration surfaces the specification implies. Mermaid C4 context and container diagrams are produced at this step, expressing the architecture in the structural vocabulary the artifact grammar names. The diagram is not an illustration of the architecture; it is the architecture at a level of abstraction the team and the AI can both read without ambiguity.
3. **Architectural constitution** (`CLAUDE.md` / equivalent), the operative grammar extracted from the architecture: the rules an agent must read before any implementation session begins. This document is derived mechanically from the architecture and the functional specification; ForgeCraft-MCP automates a substantial portion of this derivation.
4. **Architecture Decision Records (ADRs)**, one per non-obvious architectural choice, each recording the alternatives considered, the criteria applied, and the reasoning for the decision taken. ADRs are written immediately after the constitution, not reconstructed after the fact, because the reasoning is present now and will not be recoverable later.
5. **Use cases, sequence diagrams, and state machines**, the behavioral contracts between components, specified with enough precision that each diagram is simultaneously a test specification. A Mermaid sequence diagram naming a payment flow generates both the service interface contract and the acceptance test skeleton. A state machine diagram for a subscription entity enumerates valid states and valid transitions, and any implementation that permits an unlisted transition is wrong by the specification.

The cascade closes when a stateless agent given these five artifact sets can derive any valid implementation state without further human direction. That is the derivability criterion of §4.3, and it is the test the practitioner should apply before calling the specification complete. Generating diagrams after the code is written is documentation; generating them in this order is the specification act itself.

6.3 The Prompt-Bound Roadmap

The roadmap is not planned separately from the artifacts — it is derived from them. The AI reads the functional specification, architecture document, and ADRs, and produces a phased plan with a pre-generated agent prompt for each item. The binding is the operative detail: a prompt-bound item is an independent execution unit containing the relevant specification references, acceptance criteria, and verification steps. At execution time, the practitioner triggers the item and reviews the output. The result: (1) the git log is the roadmap execution record, each commit traceable to a roadmap item; (2) loop separation by granularity is enforced — short loops have fixed scope, long loops have milestone-level granularity, neither collapses into the other; (3) waiting states constitute productive inventory — a project at a natural boundary is placed in a ready state with context intact, no reconstitution required.

6.4 The Incremental Cascade

Incremental changes propagate bottom-up: the practitioner observes a discrepancy or new requirement; the AI performs an impact assessment (which artifacts reference the changed element, which roadmap items share a dependency); the cascade propagates upward to the minimum affected layers, then back down to implementation. Not every increment walks all five initialization steps. The ordering constraint is: when a layer needs updating, all layers above it are made consistent before any layer below it receives the change. The spec is always the system of record — code wrong relative to the new spec is a derivation gap, not a bug.

6.5 Loop Types and Gate Conditions

The methodology operates at four loop granularities:

- **Initialization loop** (once per project): gate = derivability criterion — a stateless agent given the complete artifact set can derive any valid implementation state.
- **Incremental short loop** (per roadmap item or spec delta): gate = full test suite passes, feature exercised at the HTTP or CLI boundary, documentation cascade complete, Status.md updated.
- **Pre-release loop** (before each environment promotion): gate = release candidate criteria stated in the test architecture document. Required: full mutation testing (pre-deployment), smoke tests across all surfaces, load tests naming the target concurrent user population and p99 latency ceiling, stress tests to failure with documented recovery, dynamic security analysis against the deployed environment. Canary and blue-green rollouts must name the canary population size, error rate rollback threshold, and observation window.
- **Hotfix loop**: minimal targeted fix ships first; post-mortem ADR and cascade artifacts follow immediately after stabilization.

Each milestone is a **phase collapse**: planning, implementation, testing, review, and deploy executed within a single session, converging because the specification holds the full intent and quality gates close the loop before the session ends. A complete practitioner's protocol is in the companion execution guide (`GenerativeSpecification_PractitionerProtocol.md`).

6.6 The Test Architecture as a Specification Artifact

The test suite is a first-class artifact, not supplementary documentation. It specifies observable behavior across every layer, couples to a commit-boundary discipline, and is generated by the AI from the project specification. A project without a stated test architecture has left the verification surface implicit — structurally the same error as leaving the system architecture implicit.

Three executor tiers. The methodology adds an orthogonal dimension to the standard test taxonomy: *who executes and against what environment*. The **automated test suite** covers the full established canon — unit through E2E, executed by CI pipelines. **Synthetic QA** is new: a capable AI agent operating in the staging or live environment reads rendered output, console errors, network traces, and stack traces in a single pass, completing in minutes what QA exploratory testing took hours to surface. **Human QA** is the irreducible judgment layer — aesthetic evaluation and ergonomic assessment — but its scope is shrinking as Synthetic QA expands. The full taxonomy, cross-referenced to commit boundary and ForgeCraft project tag, is in the companion execution guide (§§21–23).

The expose-store-to-window technique. In the test environment, the application state store is exposed to `window`. Playwright can then assert what the application believes is true — not only what renders — catching the class of failure that displays correctly but corrupts internal state.

The vertical chain test. A single UI action is traced through the service layer response, database state, and affected indexes, then back to the visible outcome. One trigger, inspected at every boundary it crosses.

Mutation testing performs an adversarial audit. An AI-generated suite may be written to pass the correct implementation rather than to catch violations of it. Mutation testing closes this gap (Jia & Harman, 2011): by introducing deliberate faults and verifying the suite detects each, it proves detection capability. Coverage measures what was executed. Mutation score measures what was caught. The second is the meaningful metric.

Three-stage multimodal quality gates. Generative asset pipelines require gates standard frameworks were not designed to address. The Shattered Stars case (§7.6) established a three-stage structure generalizable across media — visual assets, audio, and generated code:

- *Stage 1* — programmatic geometry or syntax checks (free, milliseconds): objective library-level assertions. Failure = new seed or adjusted parameters.

- *Stage 2* — composition or architecture analysis (free, seconds): structural properties without a learned model. Failure = parameter adjustment before re-run.
- *Stage 3* — vision, audio, or code model evaluation (~\$0.01/asset): semantic properties detectable only through learned understanding. Failure = structured critique injected into the next generation prompt.

Stages 1–2 filter obvious failures before the expensive evaluation runs. Stage 3 closes the gap between technical compliance and perceptual correctness. The practitioner specifies Stage 3 acceptance criteria before generation begins; the pipeline converges autonomously.

6.7 Use Cases, Diagrams, and Living Documentation

A use case in a generative specification is a multi-purpose production rule, not a requirements artifact superseded by implementation. One precise interaction description seeds three independent outputs: the **implementation contract** (actor, precondition, trigger, postcondition — what the service method is written against); the **acceptance test** (the same artifact transcribed into executable form; test difficulty is a diagnostic for underspecified use cases); and the **user documentation** (the same content with a different framing — a rendering pass, not a writing pass).

Diagram types constitute grammar layers. C4 covers static structure. The temporal and behavioral complement is: sequence diagrams fixing inter-component protocol; state machine diagrams enumerating valid states and transitions; user flow diagrams specifying the expected path. These are not illustrations — they are constraints the AI reads before generating implementations. A sequence diagram specifying authorization-before-fetch is a stricter constraint than prose, because it is unambiguous about order.

Living documentation derives from the specification. OpenAPI from TypeScript decorators, TypeDoc from JSDoc, changelog from typed commit history — documentation is a derivation from the same source as the code and cannot be wrong in a way the code is right. When the specification layer is complete, inline comments collapse: naming conventions carry semantic signal; ADRs hold decision rationale; use cases hold behavioral intent. A comment that explains *why* is a gap in the ADR record. A comment that explains *what* is a gap in the naming.

A closing observation. The methodology has two separable layers. The process layer — seven specification properties, artifact grammar, commit discipline, tooling — is community-convergent: it can be standardized, automated, and handed off. ForgeCraft automates parts of it; the community will extend it. The specification layer — domain understanding, the ability to name the correct dimension before artifacts are written — cannot be automated. It is the input the process acts on. If the process layer reaches community-maintained solved state, outcome variance becomes a function of specification quality alone. Tool mastery and syntax fluency approach zero as differentiators. What remains is the engineer's understanding of the problem — which is precisely what the process cannot supply, and what it was never designed to. The structural implications are taken up in §10.

7. Empirical Case Studies

The evidence in this section falls into two distinct categories with different epistemic weights. The six production projects (§7.1–§7.7) are the substrate from which the methodology was derived: real systems built or refactored under increasingly rigorous GS discipline, with the author as both practitioner and evaluator. They are existence proofs and diagnostic instruments — each project surfaced a failure mode, named it, and produced a corrective property. The seven-property rubric is their residue, not their premise. The controlled experiments (§7.8 — AX adversarial series, RX replication, BX blind review, EX executable sprint, DX practitioner study) are the testing phase: conducted after the methodology stabilized, with prospectively committed evaluation criteria, designed to falsify rather than illustrate. The Conduit EX experiment (§7.8.D) is the most complete single-project demonstration to date — full T1–T4 proof in a live production environment. The DX practitioner study (§7.8.A) is the first external replication across independent practitioners. These two categories must be read differently: the six projects are how GS was developed; the controlled experiments are how it was tested.

Six projects across five distinct challenge types document the method's development and demonstrate that Generative Specification generalizes beyond any single case. Each represents a fundamentally different condition the methodology must address: inheriting unknown foreign code, extending a live system without architectural structure, building from nothing at a simple scale, building from nothing at a distributed system scale, extending an existing system's domain intelligence, and migrating a broken implementation to a new platform while establishing original IP. I executed all of them with AI assistance. The practitioner's relationship to the AI changed across this arc: early projects involved sustained dialogue to establish conventions and resolve ambiguities; later projects required only course corrections and extensions, as the specification discipline internalized and the AI's role shifted from collaborator to executor. That trajectory is itself evidence that GS expertise accumulates — a point developed further in §8.13.

A structural observation applicable across the cases: in both the SafetyCorePro takeover and the BRAD migration, the pre-methodology state was itself produced through AI-assisted development, same tool class, same systems, no architectural constitution. The technology did not change between the before and after states. The specification did. This is not a designed control condition, but it is a natural one whose evidential weight is developed in §7.8.

7.1 Takeover. SafetyCorePro

Production occupational safety management system (SafetyCorePro) refactored from monolithic Next.js to a fully layered, SOLID-compliant architecture. One engineer, three sessions (Feb 14–16, 2026), zero application code lines written by the human. The pre-refactor codebase was produced through unstructured AI prompting — same model class. The comparison is AI without specification against AI with one; the specification is the independent variable.

7.1.1 Pre-Refactor State

The system prior to the refactor exhibited the following characteristics:

- 71 direct database calls from the UI layer (Prisma invocations in route handlers and React components)
- Zero unit or integration tests (23 end-to-end Playwright tests only)
- 227 `console.log` statements used as the logging infrastructure across server-side code
- Business logic distributed across route handlers with no service layer
- A single critical API route performing 100 database queries per page load
- No error hierarchy, bare `throw new Error('something went wrong')` throughout
- 17 missing foreign key indexes across 10 database models
- 126 prior commits built across two external developer identities, establishing the system as a real production codebase before the methodology engineer first touched the repository

7.1.2 The Specification

Before any implementation work began, an architectural constitution was produced: a `CLAUDE.md` file defining the target architecture, quality gates, coding standards, naming conventions, and explicit constraints. The specification was produced collaboratively with the AI assistant and established:

- Target layered architecture: UI → Service → Repository → Database, with explicit dependency rules
- Test coverage threshold: 80% minimum, enforced on every commit
- Error handling standard: custom exception hierarchy, every error carrying an ID, timestamp, and context
- Logging standard: structured logger with level gating and PII redaction
- Naming conventions, function length limits, file length limits
- Explicit forbidden patterns: no direct database calls from route handlers, no hardcoded secrets, no bare exception throws

7.1.3 Results

Metric	Value
Wall-clock time	37.5 hours (includes overnight)
Active development time	8–10 hours across 3 sessions
Commits	10 (atomic, conventional)
Files changed	174
Lines added	16,229
Lines removed	1,889
New test cases	484
New test files	27
Test lines of code	4,898
Direct DB calls removed from UI	71
Repository interfaces introduced	3 (fully swappable)
Custom error types introduced	8
<code>console.log</code> calls replaced	227
Structured logger calls added	263
DB queries per page load (critical route)	100 → 15
Missing FK indexes added	17

The test progression across commits: 75 → 97 → 218 → 285 → 339 → 346 → 484. Architecture materialized progressively. Each commit was independently valid, tested, and deployable.

7.1.4 The Significance

One instruction: *"Make it production-grade and maintainable. One atomic commit at a time."* The 16,229 lines decompose honestly: ~3,040 dependency metadata and spec artifacts; ~4,840 test lines (zero pre-existed); ~1,200 business logic redistributed into service/repository layers; ~7,150 genuinely new production code (repository interfaces, service layer, error hierarchy, structured logger, RAG/BM25 infrastructure). The headline is the structural transformation: 484 tests from zero, 71 direct DB calls removed, critical route from 100 to 15 queries — one weekend, one engineer, foreign codebase. Two acts constituted the human contribution: authoring the specification, and issuing one instruction. Pre-refactor state preserved at <https://github.com/jghiringhelli/scp-gs-experiment> for reviewer comparison.

7.2 Brownfield. Invellum

Domain: Entrepreneurial ecosystem platform, a social network for founders and entrepreneurs. Connections, social feed, project and campaign management, real-time chat, discovery, notifications, and an administrative console.

Beginning development in June 2025 (eight months before the structured restart), Invellum used earlier AI models as development tools throughout that period, but without a specification to read against, their output had no architectural home. By the time of the methodology intervention, the system had a Next.js frontend, an

Express/TypeScript API, and a Prisma-backed schema covering the major domain surfaces: authentication, profiles, connections, feed, projects, campaigns, messaging, and notifications. Working, but not extensible. No architectural discipline, no test suite, no ADRs, no layer boundaries, no specification. Eight months of informal knowledge held the system together in the engineer's memory, with no artifact form.

The challenge: Extending a live system whose structure existed only in accumulated context. The system worked; the cost was coherence without structure — every feature required reading prior work to understand where it belonged; the AI could produce output but not *coherent* output without a grammar.

The intervention: The transformation point was the introduction of Claude Opus 4.5 and ForgeCraft. ForgeCraft generated the architectural constitution (`CLAUDE.md`), covering layered architecture, SOLID standards, testing requirements, naming conventions, and explicit module boundaries. An ADR directory was introduced. A `Status.md` checkpoint file tracked session-to-session continuity. Playwright-based oracle tests defined the expected behavior of every surface before implementation resumed. The production deployment (Railway backend with PostgreSQL and Redis, Vercel frontend, environment configuration, CORS policy, and production URL validation) was executed from the CLI with Claude directing every step. The spec was not imposed on the existing code, it was written to describe what the system should become, and the system was grown into it.

The results over 36 commits (*qualitative case, no test-count baseline was established before the specification intervention; the figures below reflect the post-specification state, not a before/after comparison*):

Metric	Value
Development history	June 2025 – February 2026 (8 months) pre-specification; earlier AI models used throughout, without specification structure
Post-specification commits	36
Final production state	Live on Railway (Express backend, PostgreSQL, Redis) and Vercel (Next.js frontend)
Test coverage	17 oracle tests, 12 production smoke tests, resource audit, security audit (<i>no quantitative baseline available from pre-specification period</i>)
Security findings	Zero critical findings
Feature surface	Auth, profiles, connections, feed, projects, campaigns, chat, messages, notifications, discover, onboarding, admin console, i18n

After the specification, each feature was an implementation against a contract that already knew where it belonged. The ADRs made session decisions available as context in subsequent sessions. Invellum remains in active development at time of writing — the only ongoing case in this study. The spec does not expire when the sprint ends.

7.3 Greenfield. ForgeCraft

Domain: Developer tooling. An MCP (Model Context Protocol) server that generates production-grade AI coding assistant instruction files from a library of 112 curated template blocks. Supports six AI assistants, 19 project classification tags, and a tier system. ForgeCraft-MCP 1.0.0 is distributed freely via npm (`npm install forgecraft-mcp`). The tool is open source. The project is monetised through consulting engagements with organisations that want guided convergence cycles, bespoke quality gate authoring, or custom integration work. There is no subscription or per-seat fee.

Starting condition: A blank repository. No prior codebase, no inherited debt, no existing architecture. Pure specification-first construction.

The distinctive characteristic of this case: The tool was built using the methodology it implements. ForgeCraft generates generative specifications for other projects. It was itself built as a generative specification from day one. The `CLAUDE.md` that governed ForgeCraft's construction was structurally identical to the documents ForgeCraft would later generate for its users. The methodology was eating its own cooking from commit one.

The specification: The architectural constitution defined the MCP SDK integration contract, the template loading and rendering pipeline as a port/adaptor boundary, the tag classification system as a domain model, and the test coverage requirements. The composition root, the tool handlers, and the registry layer were all specified as interfaces before any implementation existed. Vitest was configured as the test runner with coverage gates enforced by a commit hook.

The initial release (a single commit) shipped with 14 MCP tools, 18 composable tags, 43 template files containing 112 tier-tagged blocks, and 111 tests passing across 9 test suites. There was no prototype phase, no iterative assembly toward a working state. The specification described a complete tool. The first commit delivered one. The subsequent 39 commits are documented feature additions: multi-target assistant support, the tier system, CLI mode, the MCP sentinel, domain playbooks. The breaking rename from `forgekit` to `forgecraft-mcp`, including package name, configuration format, all type names, and every documentation reference, landed in a single commit with zero test regressions. Six months of additions. Nothing revisited.

The results over 40 commits:

Metric	Value
Total commits	40
Current version	1.0.0 (released March 2026)
Tests passing	1127 (current; 111 at initial release)
Template blocks	112
Project classification tags	19
Supported AI assistants	6 (Claude, Cursor, Copilot, Windsurf, Cline, Aider)
Distribution channels	npm (<code>npm i forgecraft-mcp@latest</code>). The MCP configuration is declared in <code>forgecraft.yaml</code> ; developers register it once with their MCP client (VS Code Copilot agent mode, Claude Desktop, or any MCP-compatible host).
Breaking refactor	Full rename from <code>forgekit</code> to <code>forgecraft-mcp</code> , one commit, zero test regressions

The role of the specification: The breaking rename (package name, config format, all type names, all documentation) landed in a single commit with zero test regressions — possible because the test suite verified behavior through interfaces, not implementation structure. When the names changed, the behavior contracts held. If GS produces systems resilient to change, the proof of concept is the tool that generates the specification, built under the specification.

7.4 Greenfield (Complex). Conclave

Domain: Multi-role AI orchestration. A system that decomposes a natural language specification into a directed acyclic graph of tasks, assigns each task to a specialized AI role (Architect, Implementer, Reviewer, Tester, Deployer, Auditor), executes them in sequence with inter-role artifact flow, and provides a real-time dashboard for human oversight and gate approval.

Starting condition: A blank repository with a written specification document. The system being built was itself a system for managing AI-assisted software construction, the most structurally complex case in this study.

The challenge: Conclave is a distributed system with a monorepo architecture (9 packages, pnpm workspaces), a DAG execution engine, a message bus with bounce protocol, a rate limiter, a streaming execution layer, a React dashboard with real-time output, and a deployment pipeline with target auto-detection. Each of these is a non-trivial subsystem. Coordinating their construction without the generative specification would have required continuous human navigation of cross-package dependencies.

The specification: The architectural constitution governed the monorepo package boundaries as hard contract lines. Inter-package dependencies were explicitly mapped. Each package was given a single responsibility: core (DAG + state), actions (typed action library), roles (executor registry), dashboard-ui (React orchestration interface), MCP server (external interface), and so on. The DAG engine and message bus were specified as interfaces before any implementation; the bounce protocol and rate limiter were defined as domain models with pure behavior. A `STANDARD_PIPELINE` template with 11 phases and 15 tasks was written as a declarative configuration before any execution path was implemented.

The results over 27 commits:

Metric	Value
Total commits	27
Monorepo packages	9
Pipeline phases	11
Pipeline tasks	15
Tests	203 Vitest + 50 Playwright E2E = 253 total
Dashboard	React orchestration UI with streaming output, gate approval, retry/cancel, history
RAPTOR indexing	Hierarchical codebase summarization (file → module → subsystem → repo) injected into every task context via CodeSeeker integration (see §8.5)
Deploy detection	Auto-detects target from filesystem (Railway, Vercel, Fly, Docker)

The role of the specification: The 9-package monorepo was navigated without cross-boundary contamination in 27 commits — the AI never crossed a package boundary not defined in the architectural constitution. The 253-test suite gave continuous verification across multi-hop failure modes. Complexity is not the ceiling of the methodology; it is the argument for it.

7.5 Extension: BRAD (Legal Intelligence Engine)

Domain: Sovereign legal intelligence engine for US family law cases, live at askbrad.ai. Semantic and structural analysis of case documents: pattern recognition, argumentation logic, and deontic reasoning applied to family law proceedings. The methodology engineer arrived to extend and deepen the analytical capability of a codebase with an established external commit history. Two distinct developer identities are present in the repository.

The challenge: Like SafetyCorePro (§7.1), this case places the methodology engineer in a foreign codebase: another developer's commit history precedes the specification, and the familiarity objection (addressed in §7.8) applies least here. What distinguishes BRAD from SafetyCorePro is not that structure, it is the demand the extension placed on the specification. The question was not architectural coherence but epistemological precision: not how to extend the system, but which domains the system needed to occupy.

As with SafetyCorePro, the methodology engineer wrote zero lines of application code during the extension phase. The prior developer identity's commit history was also produced through AI-assisted development without a generative specification. The CLAUDE.md committed March 1, 2026 is the inflection point between both methodologies and both identities: what preceded it was AI output without specification; what followed was AI output with one.

The specification: The architectural constitution from the prior refactor phase was already in place. Extension work was defined as additions to that grammar: new analysis layers specified as interfaces before implementation, new domain models named with the rigor of the domain they served. The specification explicitly named the analytical techniques to be incorporated: prosody and argumentation analysis, discourse analysis, formal fallacy classification, and deontic modal logic (the formal ontology governing obligation, permission, and prohibition that structures legal reasoning). RAPTOR indexing, first specified for CodeSeeker, was carried forward in the architectural constitution and applied both to the codebase and to the legal document corpus.

The results: The analytical capability added during this extension phase included:

- **Deontic modal logic:** The formal reasoning framework governing obligation, permission, and prohibition, developed by von Wright (1951) and subsequently elaborated for legal and normative contexts, was present in the model's training corpus from academic legal theory sources and was activated by naming it explicitly in the specification. A generic "analyze arguments" prompt does not invoke this. A specification that names the domain does.
- **Discourse analysis and formal fallacy classification:** Specified as a structured analysis layer producing a taxonomy of argumentation patterns mapped to named fallacy types. The Toulmin (1958) argument model and the pragma-dialectical framework of van Eemeren and Grootendorst (2004) provide the formal vocabulary; both are part of the model's argumentation-theory training corpus and were activated by naming them in the specification.
- **AI-derived case taxonomy:** The specification directed the AI to derive the classification schema — to identify the natural orthogonal dimensions along which US family law cases differ in legally material ways. The result: a three-tier taxonomy of event types (12 tags), legal relevance tags (8 tags mapping events to statutory implications), and behavioral pattern tags (7 tags identifying relational dynamics). Jurisdiction-aware, calibrated against Minnesota family law statutes (MN 518.003, MN 518.17) and extensible across all fifty states through a jurisdiction-profile system. Practitioner review by family law attorneys remains the appropriate next step for clinical deployment.
- **Property graph knowledge layer:** A graph structure encoding case entities, relationships, and legal claims, schema, ingestion logic, and query patterns, produced from a specification of what the analysis required, not how to implement it.

Metric	Value
Developer identities in repository	2 (saxaboom, prior refactor, 7 commits; Ghiringhelli, methodology extension, 31 commits), build history at https://github.com/jghiringhelli/brad-gs-build

Specification introduced	CLAUDE.md committed March 1, 2026, marks start of extension phase
Post-specification commits	31
Files changed (post-specification)	142
Lines added	23,848
Lines removed	887
Test files	39 (37 unit + 2 e2e Playwright)
Test cases	1,183
Analysis dimensions incorporated	4 (deontic modal logic, discourse analysis, formal fallacy classification, AI-derived case taxonomy)
Knowledge structures introduced	Property graph encoding case entities, relationships, and legal claims (schema, ingestion, and query patterns)
Indexing infrastructure	RAPTOR hierarchical indexing applied to both codebase and legal document corpus

The epistemological finding: The AI's knowledge of formal legal reasoning frameworks, argumentation theory, and deontic logic is deep — it exists in the training corpus. What determines whether that knowledge is invoked is whether the specification names the domain. "Analyze legal arguments" produces legal analysis. Naming deontic modal logic, argumentation theory, and formal fallacy classification produces a specialist instrument calibrated to each field.

The author names this **domain dimensional expansion** (observed LLM behavior, coined here, with structural parallels to semantic priming, Meyer and Schvaneveldt, 1971): a domain name in the specification functions not as a keyword but as a coordinate — not a retrieval cue but a calibration signal. The model's response is not a definition of the term; it is the full apparatus of the named field at specialist depth. The pattern holds for engineering concepts equally: "upload guard" named in the spec produced a complete upload validation architecture without line-by-line specification. The concept was the specification; the architecture was its consequence.

7.6 Migration. Shattered Stars (x-wing-arcade → TypeScript/Phaser 3)

GS methodology version: v0, pre-experiment series, before the adversarial runs that produced the seven-property rubric, Known Type Pitfalls, infrastructure-first prompting, or the verify loop. Readers should interpret the results in that context: this is the methodology in its earliest form, applied to a demanding migration.

Domain: Tactical arcade space combat game. Five asymmetric factions with distinct playstyle philosophies, 100 ship types (20 per faction), ships within each faction share a coherent aesthetic and color schema that distinguishes them visually from other factions, full ability and upgrade systems, arc-based targeting, maneuver dials, AI opponents with faction-specific behavior, headless simulation for balance testing, and an AI-generated art pipeline via Stable Diffusion. Original IP.

Source system and IP origin: A Unity/C# implementation ("x-wing-arcade"): 108 commits of built gameplay drawn from the mechanical foundations of a well-known tabletop miniatures game (readers familiar with FFG's *X-Wing Miniatures* will recognize the arc-based targeting, maneuver dials, and dice-based combat resolution). The

specification was complete. The code was substantially implemented. The execution was deeply broken: runtime defects had accumulated across movement resolution, combat state, and scene management to a state where the game did not run correctly.

Current state (March 2026): See milestone table below. Playable game; campaign mode, additional modes, and balance simulation in active development.

A central technique enabling this progress is a visual execution loop that the browser-based runtime makes possible. The AI executes the game step by step, reads the browser console log output, takes screenshots and analyzes them using Claude's vision capability, sends keyboard and mouse input, and iterates, confirming that sprites are rendered and animated correctly, that the targeting arcs track as specified, and that faction AI behaves according to its behavioral template. This closes exactly the feedback loop the paper's prior tooling discussion identified as the ceiling for Unity-based AI development: the tight read-run-observe cycle that terminal-based AI could not perform. In a browser runtime, it can. The technique is not specific to games. Any system with a visual output surface and readable log channels becomes an inspection and correction target for this loop.

The migration was also driven by a tooling gap: Claude's integration with Unity was limited at the time, and the MCP ecosystem that now enables deeper editor interaction did not yet exist. The practical ceiling for AI-assisted Unity development was low for exactly the class of defects that had accumulated (runtime behavior, physics edge cases, scene lifecycle), which require the kind of tight read-run-observe loop that terminal-based AI tooling handles poorly in a Unity context.

The migration served two purposes simultaneously. **Platform reach:** Unity targets desktop builds; the intended delivery is a web-hosted static application deployable to Netlify, Vercel, or itch.io with no installation. **Original IP:** A game built on borrowed mechanics is a prototype. At migration time, Shattered Stars did not yet have a name, factions, ships, lore, or visual identity.

The specification step was also an extraction. The behavioral contracts of the Unity implementation (what each system did, not how it did it) were pulled from the existing codebase and expressed in platform-independent form. The broken execution became irrelevant. What the Unity codebase contained was a complete specification of game behavior. That specification was extracted, cleaned of any Unity API reference, and the new implementation was written against it. A broken implementation is, structurally, a complete spec with a bad executor. The methodology replaces the executor.

The specification step, before rewriting a single TypeScript file:

The methodology's claim in a migration context is that the behavioral contracts of the source system can be extracted and expressed as a platform-independent specification, one that describes *what* the system does without referring to Unity, C#, MonoBehaviour, or any API the new stack will not have. That specification then becomes the authoritative grammar against which the new implementation is written.

The output of this step for Shattered Stars:

Artifact	Content	Scale
specs/TECH_SPEC_AND_ROADMAP.md	Full platform-independent architecture: tech stack decision with comparison tables, all game systems formally specified, AI system design, rendering pipeline, balance simulation framework, risk register	2,277 lines
specs/ directory	25 supporting documents covering faction mechanics, ship stat distributions, maneuver dial definitions, UI wireframes, art generation pipeline, sound/music assets, progression systems, special ability distributions, crew point costs, point budget tables	25 files

DEVELOPMENT_PROMPTS.md	Session-scoped implementation prompts, one self-contained prompt per subsystem, ordered by dependency, each supplying the exact contract the AI session needs to implement that system without carrying forward context from prior sessions	1,496 lines
CLAUDE.md	Architectural constitution for the new stack: TypeScript strict mode, Phaser 3 patterns, commit policy, key system inventory	Project root

None of these documents mentions `MonoBehaviour` , `GameObject` , `SerializeField` , or any Unity API. The spec describes faction playstyle ("swarm tactics, expendable; shared targeting data"), maneuver resolution ("conversion of maneuver + current position/angle into target position/angle using Bezier curve path points"), and combat properties ("dice rolling, damage resolution, stress tracking"). The platform changed. The behavioral contracts did not.

Timeline: Specification written in a focused session (`TECH_SPEC_AND_ROADMAP.md` , 25 supporting spec files, `DEVELOPMENT_PROMPTS.md` , `CLAUDE.md`). Each subsystem prompt in `DEVELOPMENT_PROMPTS.md` was fed to an AI session; the engineer initiated, reviewed, and moved to the next prompt without co-authoring code. All 64 source files committed in a single batch March 7, 2026; art validation March 8. Active engagement at two points: writing the spec, and returning for the art pipeline.

Implementation results:

Metric	Value
Source files (src/)	64 TypeScript files
Total source lines	32,470
Game systems implemented	16 (CombatSystem, ActionsSystem, TargetingSystem, MovementSystem, OrdnanceSystem, TurretSystem, PilotAbilities, UpgradeSystem, SquadBuilder, FlightControlSystem, PerformanceSystem, FactionAbilities, ManeuverTemplates, TurnSystem, TurnManager, CampaignSaveManager)
Additional modules	AI system (utility-based, 5 faction personalities × 4 difficulty levels), procedural audio engine (Web Audio API), rendering pipeline, headless balance simulation framework, 8 scenes, full UI layer
Test files	17 (435 individual test cases)

Milestone state (March 2026):

Milestone	Status
Core Systems. Movement, Combat, Targeting, Actions, Turn	✅ Functional
AI System, behavior templates, closest-enemy distance + angle, faction-specific	✅ Functional
Ship Definitions: 100 ships, 20 per faction, individual style and color schema	✅ Complete
Arcade Controls, real-time gameplay	✅ Functional
Menus & UI, main menu, settings, credits, lobby, pause, game over	✅ Complete

Audio, sounds from free sources	✅ Integrated
Pre-built Squads	✅ Complete
Image Generator Service. Stable Diffusion API integration	✅ Pipeline built
Visual Execution Loop. Claude Vision + console log + input cycle	✅ Active development technique
Campaign Mode, full lore history, AI-generated portraits	🔄 In progress
Additional Game Modes	🔄 In progress
Balance & Testing, headless simulation framework	🔄 In progress

The migration extracted a complete behavioral specification from a broken Unity codebase and drove autonomous generation of 64 TypeScript files across 16 game systems — producing a playable game with 100 ships, functional AI opponents, complete menus, and integrated audio.

Longitudinal note. Executed at GS v0 — before the seven-property rubric, Known Type Pitfalls, or the verify loop. A game of this scope reaching a playable state under an early, unvalidated methodology is the finding. The project will be presented as a longitudinal case when complete: same codebase, same engineer, tracked from GS v0 through current methodology, with ForgeCraft as the active tooling throughout.

On commit discipline: The git history is thin — two commits capturing the after-state, not the construction sequence. ForgeCraft at that version generated the architectural constitution and session prompts but did not initialize a git remote; the gap was discovered at push time and has since been addressed. The structural finding: the specification held behavioral contracts and architectural coherence across sessions without the commit corpus. What the audit trail enables is *context recovery* — the spec is not a substitute for commit discipline, it is what survives when commit discipline is not applied.

The three sub-sections below show what the methodology executes across surfaces once a specification exists: environment configuration, generative art production, and automated visual QA. Each follows the same structure as the code work above.

7.6.1 Environment Setup as a Specification Problem

The desired state — a GPU-accelerated Stable Diffusion instance ready to accept API calls — was specified and handed to the AI. After several cycles of dependency diagnosis (version conflicts, CUDA gaps, driver mismatches), it identified a pre-packaged distribution with known working configurations, installed it, and verified the service was responding. A follow-up prompt to optimize throughput led it to identify batch size and VAE precision mistuned for available VRAM and adjust both. Desired state + acceptance criteria + agent iteration. The medium differs from code; the structure does not.

7.6.2 Automated Art Validation

Ship sprites (100 total, 20 per faction) are generated via Stable Diffusion with a precise contract: top-down orthographic view, vertically symmetric, pure black background. Prompt engineering (positive/negative framing) reduced rejection rates but not to zero; manual review at scale is not a pipeline. The solution: specify acceptance criteria as executable validation. `scripts/generate_sprites.py` implements four checks before a sprite is accepted:

Check	Mechanism	Threshold
-------	-----------	-----------

Vertical symmetry	Compare pixel-level left and right halves after horizontal flip; normalized 0–1 similarity	≥ 0.85
Clean background	Measure ratio of non-black pixels in the 20-pixel border region	≤ 0.30
Vertical orientation	Principal component analysis on the ship's pixel mass; extract angle of principal axis from vertical	$\leq 15^\circ$
Centering	Center-of-mass offset from image center	Informational

A sprite that fails any check triggers regeneration, up to three retries per ship. Ships that pass are logged to a preservation list so re-runs skip them. The symmetry threshold was tightened from 0.70 to 0.85 mid-project after reviewing the first batch, the initial threshold accepted sprites that were technically symmetric but visually lopsided.

The evolved architecture: staged convergence. The flat four-check baseline performs all checks at the same cost tier. The evolved design is cost-stratified: Stage 1 (geometry — symmetry, background, orientation, color histogram) is free and runs in milliseconds; Stage 2 (composition — centering, edge density, aspect ratio) is free and runs in seconds; Stage 3 (vision model evaluation — style consistency, archetype match, quality ranking against accepted sprites) costs ~\$0.01/image. Each stage failure drives a structurally different corrective action: seed randomness (Stage 1), generation parameters (Stage 2), or prompt content with the vision model's specific critique injected (Stage 3). Most rejections fail before Stage 3. The pipeline converges autonomously. The human's role is to define Stage 3 acceptance criteria before generation begins.

The pattern: desired output specified as a measurable contract at three distinct cost/abstraction levels, each driving its own corrective feedback. The same logic that governs whether a TypeScript module satisfies an interface governs whether a sprite satisfies its visual requirements. The artifact type differs. The underlying principle is identical.

7.6.3 Visual QA via Screenshot Analysis

The bug-squashing phase introduced a fourth pattern. Playwright runs against the live game and captures screenshots at defined interaction points: game states, combat sequences, UI transitions. These screenshots are passed to the AI, which reads them visually and identifies defects, misaligned UI elements, incorrect game state rendering, ships in positions they should not occupy. The defect description is then fed back as a fix prompt.

This is manual QA operationalized. The human role in a traditional QA cycle is to run the game, observe the visual output, identify what is wrong, and report it. That loop is now closed by the AI reading the screenshot. The Playwright harness provides repeatability; the visual analysis provides the judgment that a test assertion cannot, because some classes of defect are only visible, not textually detectable. The game iterates toward correctness the same way the art pipeline does: specify the acceptable state, observe the actual state, close the gap.

A more demanding variant closes the full vertical slice. A Playwright interaction fires a UI action; the AI then queries the service layer response, the database state, and any affected indexes, verifying that the effect propagated correctly through every layer, then returns to the UI to confirm the visible outcome matches the stored state. This is not a unit test and not a visual check. It is a chain verification: one trigger, observed at every boundary it crosses. A defect anywhere in the chain (service logic, persistence, index consistency, UI rendering) is surfaced in a single pass. The specification defines what the chain should produce at each layer; the AI runs the chain and reports where the actual state diverges from the specified one.

7.7 Regulated Multi-Layer Data Platform (T3 + T4)

Challenge category: Regulated cloud ETL platform with specification-governed infrastructure and self-healing diagnostic governance

Context: A public-sector data platform processes documents from multiple heterogeneous sources — court records, clinical systems, administrative data — through a medallion architecture: Bronze (immutable raw storage) → Silver (structured, extracted, validated) → Gold (resolved, patient-centric, analytics-ready). The platform integrates polyglot persistence (S3, DynamoDB, Neptune graph, OpenSearch) across four access patterns and processes sensitive data under compliance requirements. No code is shared; the architecture description is documented here as the evidence.

7.7.1 T3: Specification-Derived Infrastructure

COMPASS is a configuration-driven pipeline: CDK reads a single YAML specification and derives the entire cloud environment (S3, DynamoDB, Neptune, OpenSearch, Lambda, Step Functions, EventBridge). A mandatory NFR layer — `CompassNfrAspect` applied at synthesis time — cannot be disabled: all DynamoDB tables receive CDC streams, all Lambdas receive X-Ray tracing, all resources receive required tags. Deployments can configure thresholds but cannot remove obligations. Compliance policy (HIPAA, audit log retention) is enforced as a synthesis-time type-level constraint, not a launch checklist. The practitioner issues no CLI commands, edits no infrastructure files.

7.7.2 T4: Specification-Governed Self-Healing

The diagnostic agent — "The Eye" — has read access to the complete runtime specification of the system: the lineage graph (DynamoDB), the processing manifest (per-document version history), CloudWatch logs and metrics, X-Ray distributed traces, GitHub commit history, and CDK blueprint versions. It has no access to PII values, credentials, or file contents — only to metadata, hashes, scores, and version information. This access boundary is specified, not enforced by convention.

The formal substrate that makes diagnosis possible is the **process hash**:

```
process_hash = sha256(bronze_file_hash | script_version | config_version | llm_prompt_hash)
```

This is not a performance optimization. It is a formal invariant: the complete specification of what valid processed state looks like for a given document at a given pipeline version. If the bronze source changes, the script version changes, the configuration changes, or the LLM prompt changes, the hash changes — and the specification declares the existing downstream state invalid. The pipeline is designed so that the practitioner never manually identifies which documents need reprocessing. The hash tells the system, by formal contract, which states are stale.

The lineage graph stores the process hash, bronze hash, script version, config version, and prompt hash for every transformation event — a complete specification of every state every document has passed through. When a problem is reported, The Eye traverses this graph backwards from the reported symptom, comparing actual state to the specification at each layer, and stops when it finds a discrepancy. It does not scan the full system. It reads the formal record and halts at the first specification violation.

Root cause taxonomy: seven types (low confidence, stale data, code bug correlated with git commit, config change, source corruption, missing rule, human-review edge case). Data-driven failures trigger autonomous reprocessing; code-driven failures generate a structured GitHub issue and await deployment approval. Safe actions execute without approval; escalation actions and DataZone governance approval gate any output change — the human governance role preserved at the boundary the specification has not yet closed. The practitioner is released from: diagnosis, root cause investigation, document reprocessing identification, and correction decisions for known failure types.

7.7.3 The Formal Connection Between T3 and T4

The COMPASS case demonstrates something the six prior cases cannot: the dependency between tiers. T4 diagnosis is only possible because T3 captures the complete provenance of every infrastructure and processing decision. The process hash is a formal contract that propagates from the specification downward through every pipeline stage. The lineage graph is not logging; it is the runtime specification of what valid state looks like at every step. When the diagnostic agent reads the lineage graph, it is reading the specification of what should have happened and

comparing it to what did happen. The correction mechanism derives from the same source as the construction mechanism, which is the T4 claim stated precisely.

The case also demonstrates the restriction/liberation duality operating across both tiers simultaneously: the mandatory NFR layer (T3 restriction) is what makes proactive monitoring and compliance audit (T4 capability) possible without practitioner intervention. A system whose infrastructure is not fully governed by specification cannot have its runtime behavior evaluated against that specification. T3 completeness is the precondition for T4 correctness.

7.8 Threats to Validity and Experimental Closure

The Define/Build/Measure Loop

The primary methodological challenge is not any single threat, it is a structural one that runs through the entire experiment series. GS defines the seven specification properties. GS guides the AI to produce implementations that satisfy those properties. GS scores whether the implementations are good. A discipline that defines "good," builds toward it, and then measures whether it achieved it has a circularity problem that no amount of external checking fully eliminates. This is the load-bearing concern. Every other threat in this section is secondary to it.

The loop has three distinct layers, each requiring a separate closure mechanism:

Layer	Threat	Closure Mechanism	Status
3, Output measurement	External checks use criteria the rubric author defined	<code>tsc --noEmit</code> , ESLint (<code>eslint:recommended + @typescript-eslint/recommended</code>), <code>npm audit</code> (supply-chain gate, not code quality), and the Conduit test suite, 104 tests authored by the open-source community, not this paper's author	Closed
1, Rubric validity	Rubric rewards GS compliance, not objective quality	BX: three Conduit implementations never built with GS scored blind, rubric ranking (13/14 > 7/14 > 6/14) congruent with CVE count, test count, and TypeScript health on every axis	Closed
2, Guidance circularity	GS guided the implementation AND scored it	DX (§7.8.A): blind evaluator, dual rubric (GS properties + external structural battery), 58 participants (83 analyzable submissions) across two conditions	Closed — §7.8.A.1

The ordering is deliberate: Layer 3 is the weakest mitigation (the tools are independent but the implementations were still GS-guided); Layer 1 is stronger (the rubric is applied to implementations it never guided); Layer 2 is the complete closure (the guidance-plus-measurement loop is broken by human participants and a blind evaluator). All three layers are now closed. Layer 2 results are in §7.8.A.1.

One finding from BX merits explicit disclosure: the Defended property reveals a persistent tooling-layer limitation. No implementation across AX, BX, or RX scores 2/2 on Defended. A CI pipeline can be specified in a GS document; it cannot be provisioned by generated code. CI runners require external infrastructure that AI-assisted generation cannot physically create. This is not a gap in the methodology: the specification correctly describes what must exist, the convergence loop requires that nothing important is deployed before it runs, and for safety-critical domains the methodology explicitly calls for human review on top. The 0/2 score is an honest record of what the current generation of AI tooling can deliver, not a bound on the methodology's claim. It is documented here rather than scored aspirationally.

Additional Threats

Threat: Single-engineer design introduces selection bias. *Assessment:* The codebase-familiarity objection does not survive the evidence — SafetyCorePro and BRAD both carry forensically distinct prior contributor identities; the methodology engineer arrived as an outsider to foreign code. The specification-authorship confound remains: every specification was written by the methodology's creator. This is the specific limitation DX (§7.8.A) is designed to address; results are in §7.8.A.1. The methodology establishes a floor; the ceiling is set by the practitioner's domain depth. Independent replication will quantify how much the floor alone accomplishes.

Threat: No control condition. *Assessment:* SafetyCorePro and BRAD both provide natural controls: the pre-methodology codebase was itself produced through unstructured AI prompting, same model class. Technology was constant across before/after states; the specification was not. The structural differences (zero tests → hundreds, monolithic → layered) cannot be attributed to the technology. Before/after states are preserved in version control and available to reviewers on request.

Threat: Self-referential rubric. *Assessment:* Substantially true; mitigated at three levels matching the loop table above. Layer 3: external checks (`tsc` , `ESLint` , `npm audit` , the open-source Conduit 104-test suite) are rubric-independent and directionally consistent with GS rubric scores. Layer 1: BX scores three RealWorld implementations without GS exposure; rubric ranking ($13/14 > 7/14 > 6/14$) is congruent with CVE count and test coverage on all axes. Layer 2: DX (§7.8.A) uses a blind evaluator and a dual rubric where one battery predates GS. Neither mitigates the primary circularity in the GS audit scores; the limitation is named honestly, and DX is its designed resolution.

Threat: Self-reported metrics. *Assessment:* Core claims (commits, files, lines, test counts, layer boundary conformance) derive from git history, reproducible from repository access available on request. Multi-author attribution is forensically verifiable from commit log. The one genuinely self-reported metric is active development hours; commit timestamps confirm plausibility but do not verify precisely.

Threat: Incomplete tooling confounds results. *Assessment:* This is a directional argument, not a threat. If incomplete tooling produced these outcomes, complete tooling establishes a floor: the results are a lower bound on the methodology's potential.

Threat: AX ran on a single model. *Assessment:* All AX conditions were executed on claude-sonnet-4-5. Cross-model generalizability is untested. The expected direction on any instruction-following model is positive: the value GS adds is structural (artifact completeness, boundary explicitness, decision persistence), not prompt phrasing. A more capable model should require less scaffolding; a less capable model may require more. The magnitude of the effect is model-dependent; the direction should be model-independent. Independent replication on alternative models is invited.

Independent replication is invited. Experimental conditions, prompts, and scoring rubric are archived at <https://github.com/jghiringhelli/generative-specification/tree/main/experiments/ax/> . It is the author's explicit intent that the methodology be tested and that it survive the test.

7.8.0 Validation Strategy

Three experiment series form a layered case. **AX** (solo layer): the methodology applied across eight conditions and measured against an external rubric — closes whether the procedure can be executed consistently, not whether it is self-referential. **BX/RX** (peer layer): three RealWorld implementations built without GS knowledge, scored blind — closes circularity at the output measurement layer using the open-source 104-test Conduit suite. **DX** (professional layer): external practitioners, blind evaluators, dual rubric where one battery predates GS — closes the guidance layer. None alone is sufficient; together they constitute a layered case.

7.8.A Experiment I: Human Practitioner Study, April 2026

Controlled practitioner study: ~70 developers, Mexico City, April 10, 2026. Two conditions: Group A (control, free prompting) and Group B (treatment, full GS artifact set — architectural constitution, ADR set, sequence diagrams, schema definitions). Evaluator blind to condition scores each output against the six-property GS rubric and an independent external battery (test coverage, cyclomatic complexity, ESLint violation rate) that predates GS. Socratic session structure: control session first, GS concept introduced through dialogue, treatment session after. Four pre-registered predictions: (P1) discipline transferable in one session; (P2) effect independent of GS rubric; (P3) control failures consistent across practitioners; (P4) without discipline, AI output is arbitrary. Pre-registration record: committed to `experiments/dx/pre-registration/` before the April 10 session; timestamp independently verifiable from the repository commit log. Results in §7.8.A.1.

Ethics and consent. Participants were informed of the study's purpose before the session began — that their outputs would be analyzed and potentially reported in aggregated, anonymized form for research. Participation was voluntary; withdrawal at any time without consequence was explicitly communicated. No personally identifiable information appears in reported results; outputs are identified only by anonymized branch identifiers and condition assignment. Consent was obtained via a Microsoft Forms pre-session declaration completed by each participant before the session began; records are retained and available to reviewers on request. The study operated within an organizational facilitation context; the consent procedure was designed to meet standards of informed voluntary participation.

7.8.A.1 DX Results: April 10, 2026 — Mitikah, Mexico City

Design as executed

58 developers, April 10, 2026, Mitikah campus. Two greenfield projects in sequence: Vaquita (cooperative payment API, morning, pre-reveal) and Taskflow (Kanban API, afternoon, post-reveal). Condition A = control (free prompting); Condition B = treatment (ForgeCraft pre-generated artifact set).

Critical design note: The two sessions are not equivalent experiments. Vaquita is a clean tool-effectiveness comparison — neither group had GS knowledge. Taskflow is a *free GS application* (A) versus *compliance with a pre-reveal artifact* (B) comparison — Condition B participants were instructed to follow the ForgeCraft roadmap generated before either group knew GS, even after gaining GS understanding during the reveal. The sessions answer different questions and are interpreted separately.

83 of 116 submissions met the analyzable threshold. Condition A: 44 submissions. Condition B: 39 submissions.

Measurement

Each of the 83 analyzable branches was processed through an automated pipeline: static complexity analysis (Lizard), live API test scoring (Hurl, 10-step test suites against running servers), and the GS rubric battery. The executable score (0–3 points, derived from Hurl pass rate) serves as the primary paradigm-independent functional completion metric.

Quantitative results

Table 1: Executable score by session and condition

Session	Condition	N	Mean	Median	SD	% Perfect (3/3)	Zero scores
Vaquita (pre-reveal, pure tool test)	A — free prompting	24	2.00	2.00	1.02	38%	3 (12%)
Vaquita (pre-reveal, pure tool test)	B — ForgeCraft	23	1.52	2.00	0.95	13%	4 (17%)

Taskflow (post-reveal, GS-informed)	A — manual GS	20	2.65	3.00	0.75	75%	1 (5%)
Taskflow (post-reveal, GS-informed)	B — ForgeCraft	16	2.13	3.00	1.31	63%	4 (25%)

Table 2: GS rubric and process metrics by condition (all sessions combined)

Metric	Condition A	Condition B	Δ
Executable score (0–3)	2.30	1.77	–0.53
GS total (0–8, 5-property workshop rubric)	6.07	6.46	+0.39
GS auditable (0–2)	1.56	2.00	+0.44
Commit count	46.5	59.9	+13.4
Test coverage (%)	74.5	73.3	≈

Table 3: Inference statistics — executable score (Mann-Whitney U, two-tailed)

Session	U	z	p	Rank-biserial r	Cohen's d	95% CI median(B–A)
Vaquita (pre-reveal)	196.5	–1.776	.076	.282 (A > B)	.485 (A > B)	[–1.50, 0.00]
Taskflow (post-reveal)	133.0	–1.059	.289	.164 (A > B)	.508 (A > B)	[–1.50, 0.00]

Vaquita high-quality proportion (score = 3): A 38% vs. B 13%, $\chi^2(1) = 3.70$, $p < .05$. Taskflow high-quality proportion: A 75% vs. B 63%, $\chi^2(1) = 0.66$, $p = .443$. Bootstrap 95% CIs estimated from 50,000 resamples; Mann-Whitney U uses normal approximation with tie correction. Per-branch score distributions archived at [experiments/dx/results/](#) ; replication script at [experiments/dx/analysis/dx1-stats.js](#) .

Statistical interpretation. The Vaquita session (pre-reveal, pure tool test) shows a marginal overall effect ($p = .076$) with a small-to-medium effect size (rank-biserial $r = .28$) and a significant difference in the proportion of perfect scores ($p < .05$). The group medians are identical at 2.00; what differs is the high end of the distribution — Condition A produced three times as many perfect-score outputs (38% vs. 13%). The Taskflow session shows no significant overall difference ($p = .289$) — consistent with the temporal mismatch confound described in Finding 2: the condition contrast is not clean (A applied GS freely; B was constrained by a pre-reveal artifact). Both sessions show group medians overlapping at 2.00 and 3.00 respectively with wide confidence intervals, reflecting the study's underpowered design. Effect sizes (Cohen's $d \approx 0.5$ in both sessions) are in the medium range and are directionally consistent, supporting the pre-registered prediction direction without establishing statistical certainty. DX2 is powered for formal inference with explicit effect size thresholds.

Interpretation: two findings, not one

Finding 1 — Vaquita (pre-reveal, pure tool effectiveness). Before any GS knowledge, ForgeCraft artifacts produced better process discipline (higher GS rubric scores, more commits, better auditable) at a cost to functional completion (13% perfect vs. 38%). The structural explanation: ForgeCraft front-loads specification investment. Practitioners who engaged fully produced well-structured but unfinished projects. This is a product gap — the pre-generation sequence produces artifacts but not scaffolded implementation prompts guiding spec-to-running-code. A

secondary finding: Condition A scored higher on Bounded than Condition B, a time-pressure artifact — ForgeCraft specifies the bounded architecture as an obligation; under pressure, some practitioners bypassed the service layer to get working endpoints. The fix is a single ESLint rule for Bounded-violation at commit time, not a methodology change.

Finding 2 — Taskflow (post-reveal, free application vs. pre-reveal artifact compliance). After the reveal, Condition A (free GS application) outperformed Condition B (ForgeCraft pre-reveal roadmap) on functional completion (75% perfect vs. 62%; mean 2.65 vs. 2.12; zero-score rate 5% vs. 25%). The critical variable is *artifact temporal mismatch*: the ForgeCraft roadmap was generated before either group knew the methodology. Condition B participants were instructed to comply with it even after gaining GS understanding; several asked whether to deviate and were told no. Condition A practitioners applied GS-informed judgment directly; their better completion reflects freedom from a stale artifact, not superior skill.

The combined picture. GS and ForgeCraft are not the same thing. The workshop measured both in different sessions. Scaffolding instills discipline in novice practitioners; it becomes a constraint for practitioners who have internalized the methodology. ForgeCraft needs a practitioner mode: once understanding is acquired, the roadmap should serve as reference, not prescription.

Pre-registered prediction outcomes

P1 (transferable in one session): *Confirmed with qualification* — Taskflow Condition A applied GS freely and outperformed ForgeCraft-guided B, but A had freedom while B had a compliance instruction. A further uncontrolled variable: whether participants self-selected into different effort or engagement modes after the reveal, in ways the condition label does not capture. The condition assignment holds pre-reveal; behavioral divergence post-reveal is not independently verifiable from the data artifacts. P2 (effect independent of rubric): *Partially confirmed* — Vaquita saw rubric/functional divergence attributable to tool time-cost. P3 (control failures consistent): *Confirmed* — same failure classes (lower auditable, fewer commits, architectural inconsistency) across practitioners. P4 (without discipline, output is arbitrary): *Confirmed in Vaquita; partially disconfirmed in Taskflow* — post-reveal Condition A showed more consistent output without tooling.

Timestamp analysis confirms reveal overlap with Taskflow session start; Condition A achieved higher perfect-score rates despite reduced coding time, strengthening the Taskflow finding. The study measures first-session exposure; Study 2 tests sustained independent application (~May 2026).

Follow-up studies

DX2 — Transfer through artifacts alone (May 2026). The DX1 temporal mismatch artifact (ForgeCraft generating artifacts before practitioners understood the project) was a product failure, not a methodology failure. DX2 corrects this by providing a pre-written, complete specification upfront and using ForgeCraft only as a gate enforcer. This decouples "can practitioners use a spec?" from "can they generate one under time pressure?" Pre-registered hypothesis: treatment condition (spec provided) will score ≥ 2 points higher than control (no spec) on the 14-point rubric, with no significant difference in functional completion. Blind scoring by two independent evaluators; inter-rater reliability measured (Cohen's κ). Results to be integrated in §7.8.A upon completion.

DX3 — Team coordination (Q3–Q4 2026). Tests whether GS discipline scales from individual practitioners to coordinated teams. Chronicle's shared memory layer (Axon) is the instrument. Pre-registered hypothesis: teams governed by shared specification + Chronicle memory + ForgeCraft-gated merge requests produce higher integrated GS scores and lower coordination overhead than teams working on the same project without shared specification governance. Results to be integrated upon completion.

Study 2 (~May 2026): structured survey + voluntary coding session to test sustained independent application without methodology mention — direct transfer evidence.

7.8.B Experiment II: Multi-Agent Adversarial Study: Results

Three findings drive everything that follows. **The primary prospective finding is narrow and should be stated plainly:** the GS v1 treatment scored 10/14 against the control's 9/14 — a one-point difference on a single benchmark, where GS's sole advantage was the Composable property, traceable directly to a SOLID clause the control did not include. On Executable specifically, the control outperformed GS v1: the control's full suite passed; GS v1's suite had 6/10 suites blocked by a JWT type narrowing pattern not yet named in the specification. A one-point prospective advantage on one benchmark is not a paradigm demonstration. It is the starting point for a diagnostic series. First: the boundary that matters is naive vs. structured — unstructured AI deployment produced an internally incoherent project with zero passing tests; both structured conditions produced compilable, layered code. Second: 14/14 on the full seven-property rubric is achievable, the post-hoc conditions demonstrate this, with the explicit caveat that treatments v2 through v6 were designed with full knowledge of each prior condition's gaps. This is iterated optimization on a single benchmark, not independent confirmation; the progression from 3→14 is the evidence that each gap is diagnosable and closable, not a statistical demonstration of convergence behavior. Third: the experiment both measured and corrected the methodology — the three template changes confirmed by treatment-v2 were committed to production templates and propagate to every GS-governed project; the gap between experimental finding and production tooling is zero.

One benchmark (Conduit), one model (claude-sonnet-4-5), one author's specifications: the AX study is N=1 on every structural axis. The empirical claim, a quality gradient observable under controlled conditions, is stated without population-level authority; the paradigm claim does not require it, being a structural argument about what specification completeness permits an executor to derive, not an effect size across benchmarks. The AX findings establish proof-of-concept for the measurability of the GS rubric and the correctness of gap diagnosis: each condition shows a diagnosable, closable gap. External static analysis tools (tsc, ESLint, madge, jscpd) independently corroborate the direction across all 8 conditions without access to the GS property definitions.

Benchmark: RealWorld (Conduit) backend API, a full-featured REST application (authentication, articles, profiles, comments, tags, favourites) implemented in TypeScript/Node.js/Express/Prisma against a live PostgreSQL database. Chosen because it has a published specification, a community Postman collection for conformance testing, and known correct implementation patterns, making automated evaluation straightforward and reviewer replication feasible. All three conditions ran on `claude-sonnet-4-5`, March 13 2026. The experiment design and scoring rubric were prospectively committed in commit `bd2c05b` before any condition was run (verified by timestamp at <https://github.com/jghiringhelli/generative-specification/tree/main/experiments/ax/>).

Eight conditions (three prospectively designed, with conditions committed to the public repository before execution, verified by timestamped commits at <https://github.com/jghiringhelli/generative-specification/tree/main/experiments/ax/>; five post-hoc):

Four agentic conditions run against a fixed repository: (1) no specification, session prompts only; (2) partial specification, architectural constitution only, no ADRs; (3) full GS as defined in this paper; (4) full GS with the paper as RAG-accessible source of truth. Each produces code, tests, and commit history. An adversarial auditor agent in a separate session — given the six properties as an independent rubric, not the paper — evaluates each artifact set. The comparison is at the artifact level: does the specification implied by the output satisfy the derivability criterion for a stateless reader?

7.8.B.1 Results

Condition	Key additions	Pre-registered
Naive	3-line README, 4-line prompts — de facto default	—
Control	Detailed README + 7 prompts averaging 30 lines (expert prompting)	✓

Treatment (GS v1)	17 GS context files (CLAUDE.md, Status.md, 4 ADRs, C4, use cases, NFR); 8-line prompts	✓
Treatment-v2	+ "Emit, Don't Reference" directives; First Response Requirements list of 9 mandatory P1 artifacts	No
Treatment-v3	+ Dependency governance (argon2 over bcrypt; npm audit gate as P1 requirement)	No
Treatment-v4	+ ADR emission precision fixes; verify loop (materialize→tsc→jest→correct, max 5 passes)	No
Treatment-v5	Dedicated <code>00-infrastructure.md</code> prompt before feature prompts; Known Type Pitfalls in CLAUDE.md	No
Treatment-v6	+ DRY gate (jscpd < 5%); Interface Completeness gate; ESLint as P1	No

GS audit scores, unified 7-property rubric (all conditions re-audited; AI auditor agent, blind session, no prior context of the authoring conditions, scale 0–2 per property; no inter-rater reliability check was conducted across human evaluators):

Here N=7 refers to the seven specification properties in the GS rubric, not to participant count; each experiment iteration tested derivation quality as a function of rubric compliance score, with six AX iterations below 14/14 and one (v7) reaching 14/14.

Property	Naive	Control	Treatment	T-v2	T-v3	T-v4	T-v5	T-v6
Self-Describing	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
Bounded	1/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
Verifiable	1/2*	2/2	2/2	2/2	2/2	2/2	2/2	2/2
Defended	0/2	0/2	0/2	2/2	2/2	1/2	2/2	2/2
Auditable	0/2	0/2	0/2	1/2	2/2	1/2	2/2	1/2
Composable	0/2	1/2	2/2	2/2	2/2	2/2	2/2	2/2
Executable	1/2‡	2/2‡	2/2‡	2/2‡	2/2‡	1/2	2/2†	2/2†
Total	3/14	9/14	10/14	13/14	14/14‡	11/14	14/14†	13/14†

* Naive Verifiable: the original audit scored this 2/2 based on test structure and naming; the unified re-audit applied a stricter behavioral criterion, tests must compile and tests must run, reducing the score to 1/2. All six naive test suites fail to compile due to missing schema models (real coverage 0%). ‡ Naive–Treatment-v3 Executable is auditor-inferred: the re-audit assessed whether generated code compiles and tests exist, not whether tests pass against a real execution environment, no verify loop ran for these conditions. † Treatment-v5 and Treatment-v6 Executable 2/2 is session-verified. Treatment-v5: the verify loop confirmed 109 total tests (independent re-run: 106 passing, 3 test-isolation failures in `article.test.ts`, duplicate user registration in a preceding test leaves token undefined in cleanup; not implementation errors) across 10/11 suites against a live PostgreSQL database, converged in 2 fix passes (four runner infrastructure bugs fixed before the verified run, see companion supplement §S9.6). The AI integration response reported 114 total tests; 109 is the runner-confirmed count and is the figure used throughout. No `jest-output.json` artifact was committed for v5 in the way that RX evidence was committed to [experiments/rx/evidence/](#); the verification was conducted within the audit session rather than as a reproducible committed artifact. This is the

remaining epistemic gap between v5 and RX. Treatment-v6: `session-summary.md` Final Results table confirms 62/62 tests passing with 0 tsc errors and 0 ESLint errors, verify loop converging in 3 fix passes. Treatment-v3 and treatment-v5 share the same score (14/14) with completely different epistemic bases: treatment-v3's score is auditor-inferred from static artifacts; treatment-v5's is session-confirmed by a passing test suite against a live database. The verify loop's value is not the score, it is the guarantee that the score reflects something real.

Treatment-v6: 13/14. Sole gap: Auditable 1/2 — Status.md absent. Executable 2/2 session-verified: 62/62 tests, 0 tsc, 0 ESLint errors. First Stryker mutation gate in `ci.yml`, not yet scored under rubric. Full per-condition justifications at `experiments/ax/treatment-v6/evaluation/scores.md`.

The progression 3→9→10→13 (out of 14) is monotonic across five post-hoc conditions on every instrument. The direction is unambiguous; the magnitude of the three prospectively-designed conditions, as single-model single-run evidence, is not.

Primary finding — naive vs. structured: Naive produced an internally incoherent project; all six test suites fail to compile (schema additions described in prose, not emitted as code blocks). Both structured conditions produced compilable, layered code. The naive→structured boundary is the finding that matters; control→treatment (one point, Composable) is a directional signal.

Defended floor — behavioral constant: All three prospectively-designed conditions scored 0/2. A GS artifact that *specifies* a hook does not cause the hook to *exist* — models treat specification text as guidance for application code, not as directives for operational infrastructure. The fix: fenced file templates in a First Response Requirements list cause emission. Defended moved 0→2 in treatment-v2. Failure was instruction precision, not model capability.

Treatment-v2 achieved 12/12. All three gaps (Defended 0→2, Auditable 1→2, Composable maintained) closed in a single post-hoc run. The changes were centralized: three additions to `templates/universal/instructions.yaml`, propagating to every GS-governed project. An expert-prompting practitioner who discovers the same gaps must update every project README individually. That asymmetry — one template change vs. N project changes — is the democratizable difference.

Mutation score gap: GS v1 reported 93.1% line coverage but 58.62% MSI. Line coverage measures execution; mutation score measures detection. A test that covers a line without asserting correctness passes coverage and fails mutation. After three rounds of Stryker-guided assertion improvements, the treatment project's mutation score converged to 93.10% — matching line coverage. The same gap appeared in Shattered Stars (80% line coverage, 58% MSI). When both converge, every covered line is verified. Treatment-v2 added the mutation gate as a hard P1 criterion.

What GS adds over expert prompting: Bounded (layer discipline) is achievable through expert prompting. GS's specific contributions are Composable (interface-based DI), Auditable (decision record persistence), and Defended (operational enforcement infrastructure). The democratizable difference is compounding: GS improvements cascade forward via template; expert-prompting improvements stay local.

Three template changes confirmed by treatment-v2:

1. *Mutation gate as a hard quality criterion.* MSI \geq 65% overall blocks PR merge; \geq 70% on new/changed code. Run Stryker per module after test authoring, not only at release. Propagates to all GS-governed projects on next `forgecraft refresh_project`.
2. *Emit, don't reference.* Infrastructure files, hooks, CI workflows, commitlint configuration, ADR stubs, IRepository interfaces, are named as files to be emitted in P1 with fenced templates. Treatment-v2 confirmed this change produces the artifacts; prior treatment specified them and produced none.
3. *Line coverage and mutation score are complementary, not interchangeable.* Line coverage measures execution; mutation score measures detection. A gap between them is the fraction of covered code that tests cannot verify. When they converge, every covered line is verified. Both gates are required.

Post-publication finding: architectural compliance and dependency security are fully orthogonal. Static quality checks were run across all four materialized conditions after the primary results were reported. The three checks required no running server and were applied with consistent flags across all conditions.

Check	Naive	Control	Treatment	Treatment-v2
tsc --noEmit errors	41	1	0	0
ESLint problems (bare baseline)	29	40	40	21
npm audit high CVEs	3	0	3	9

† ESLint violations measured with `eslint:recommended` + `@typescript-eslint/recommended` applied consistently across all conditions. The increase from Naive (29) to Control/Treatment (40) reflects additional source files generated in later conditions, not a degradation in per-file quality; Treatment-v2's reduction to 21 reflects the DRY and interface completeness gates specified in that condition.

Note: `tsc --noEmit` and ESLint measure static code quality, compiler correctness and style/safety rules respectively. `npm audit` measures supply-chain hygiene, known CVEs in the dependency graph. Both categories are rubric-independent; neither subsumes the other, and they are presented as separate gate categories rather than a unified quality score.

Treatment-v2, the first condition to achieve a 12/12 GS audit score, also has the highest vulnerability count: nine high CVEs, versus zero for the control. The source is a devdependency chain: `@typescript-eslint` pulling an old `minimatch` version. The control avoided this by selecting a different password library. Neither choice was architecturally motivated; both were made by the model without explicit guidance. The finding establishes that the GS rubric measures structural quality (layer discipline, interface enforcement, test construction, enforcement infrastructure) and does not assess supply-chain security. Both are necessary pre-release blockers; neither subsumes the other. A complete gate requires both an architectural audit *and* a vulnerability scan as independent checks. Full per-condition npm audit detail is in the companion supplement (§S9.3).

Post-publication condition, V3: Dependency Governance. A third post-hoc run was conducted after the static quality analysis, targeting the CVE finding directly. The condition added one prescriptive layer to the GS v2 template: explicit dependency governance instructions requiring `npm audit` to pass (zero high CVEs) as a P1 requirement, and naming preferred packages for password hashing (avoiding the `bcrypt` → `@mapbox/node-pre-gyp` → `tar` CVE chain). All other artifacts were unchanged from treatment-v2.

Property	Treatment-v2	Treatment-v3	Delta
Self-Describing (0–2)	2	2	0
Bounded (0–2)	2	2	0
Verifiable (0–2)	2	2	0
Defended (0–2)	2	2	0
Auditable (0–2)	2	1	–1
Composable (0–2)	2	2	0
Total (0–12)	12	11	–1
npm audit high CVEs	9	0	–9

Note: Scores above reflect the original direct-comparison audit at the time of the v3 run. The unified re-audit (see unified table, above) revised these scores under the full seven-property rubric and a stricter Auditable behavioural criterion: T-v2 Auditable revised to 1/2 (ADR referenced-but-not-emitted retroactively disqualified); T-v3 Auditable revised to 2/2 (dep governance directives created a richer, independently auditable trail). The comparison table is preserved as the historical record of the condition that motivated the ADR emission fix.

Principal finding: The dependency governance condition eliminated all high CVEs (9 → 0). One specification directive, prescriptive package selection and an explicit `npm audit` gate, closes the entire vulnerability surface while maintaining all other GS properties.

Auditable regression, root cause: The v3 score dropped one point from v2's perfect 12/12. The model referenced `docs/adrs/ADR-0001-stack.md` in the README and emitted `CHANGELOG.md` as a structural stub with no entries. The auditor correctly penalized both: the ADR was referenced but never emitted as a file; the CHANGELOG satisfied the presence requirement but not the content requirement. The root cause is a precision gap in the "Emit, Don't Reference" instruction: the template specified "emit ADR stubs" but did not state that each ADR must be a fenced file block with substantive content in P1, not merely cited in documentation prose and not as an empty placeholder. This is a template specification issue, not a GS architectural flaw.

Template fix applied; treatment-v5 complete, 14/14. The ADR emission precision gap was patched in `templates/universal/instructions.yaml`. Treatment-v5 achieved 14/14 by adding a dedicated `00-infrastructure.md` prompt (infrastructure must complete before feature prompts) and documenting the `jsonwebtoken` `StringValue` type pitfall in the specification. Session-verified: 109 tests across 10/11 suites against a live PostgreSQL database, converged in 2 fix passes. The root cause of prior Executable failures was a known type narrowing pattern the specification had not yet named — once named, it became a quality gate. An independent **Replication Experiment (RX)** commits `jest --json` output as a standard evidence artifact; any researcher with an Anthropic API key can reproduce the result at [experiments/rx/](#). Full supplementary data in `GS_Experiment_Supplement.md`.

Full replication data, session IDs, prompt texts, blind audit transcripts, per-condition metric tables, mutation testing progression, failed runs disclosure, and replication instructions, are in the companion supplement: `GS_Experiment_Supplement.md` (available at https://github.com/jghiringhelli/generative-specification/blob/main/docs/white-paper/GS_Experiment_Supplement.md).

7.8.C Experiment III: Benchmark Cross-validation (BX). Results

Purpose: Close Layer 1 of the define/build/measure loop. The circularity closure argument from BX is not that implementations were scored blind to condition — blind scoring cannot remove rubric self-referentiality. The closure is that rubric rankings prove congruent with external metrics the rubric never specified: CVE count, test count, and TypeScript health across implementations that received no GS guidance. If an author-designed rubric discriminates in the same direction as independent static analysis tools, the rubric is measuring something that exists outside the author's methodology.

Implementations scored:

ID	Repository	Stack	Community Signal
A	lujakob/nestjs-realworld-example-app	NestJS + TypeORM + MySQL	~2k stars; cited NestJS reference
B	gothinkster/node-express-realworld-example-app	Express + Prisma + NX	Official RealWorld benchmark

C	GS-generated RX output	Express + Prisma (GS-specified)	104/104 tests; 0 CVEs
---	------------------------	---------------------------------	-----------------------

Repos A and B were never exposed to GS methodology. Scoring was conducted blind against the rubric before comparing with external tool results.

GS Rubric Scores:

Property	Repo A (NestJS)	Repo B (Official)	Repo C (GS)
Self-Describing	1	1	2
Bounded	2	1	2
Verifiable	0	1	2
Defended	0	1	1
Auditable	1	1	2
Composable	1	1	2
Executable	1	1	2
Total	6/14	7/14	13/14

External tool alignment:

Metric	Repo A	Repo B	Repo C
tsc errors	0 (after setup)	0	0
npm audit CVEs (total)	105 (16 critical)	43 (1 critical)	0
Test cases	1	27	104 passing

Ranking congruence: Rubric order (C > B > A) is identical to CVE rank and test rank. The rubric did not require GS guidance to produce this ordering.

Principal findings:

1. *Community reputation is an unreliable quality proxy.* Repo A (2k stars) scores below the official reference (Repo B). NestJS framework discipline yields 2/2 on Bounded, the framework enforces it, while the implementation carries 105 vulnerabilities (16 critical) and 1 test case. The rubric surfaces what star count ignores.
2. *The rubric discriminates on GS-specific contributions.* Both non-GS implementations score 0/2 on Defended (no CI, no pre-commit hooks, no enforced gates) and 1/2 on Auditable (partial conventional commits, no ADRs). These are the properties with no framework analog, the AI cannot emit them from NestJS conventions alone. The GS-generated implementation achieves 2/2 on both. The rubric identifies GS's specific contribution over what a high-quality framework already provides.
3. *The Defended gap is structural, not incidental.* No implementation scores 2/2 on Defended. A CI pipeline requires external infrastructure that generated code cannot provision. This is consistent across AX, BX, and RX and is reported as a permanent limitation, not a scoring anomaly.

Full scores and per-property rationale: `experiments/bx/scores.json` .

7.9 Meta-Application: Autonomous Specification Evolution

ForgeCraft 1.0's gate system, enforcement hooks, and template hierarchy emerged from the AX experiment series — it was the output of the series, not a prior condition of the case studies. Each AX treatment cycle applied GS to ForgeCraft itself: the tool was simultaneously the specifier and the subject. The AI identified specification gaps, authored gate definitions, implemented gate logic with tests. The human contributed the rubric and the release gate. The AX series is directionally convergent with a non-monotone path (v3: 14/14 → v4 regression → v5 recovery). The same seven properties governing ForgeCraft's construction served as the rubric against which ForgeCraft's outputs were evaluated — the pragmatic tier is self-applicable. The AX self-application cycle converges to $S_{\text{realized}} = 1.0$ across the automatable rubric.

8. Implications for Practice

GS provides a structured specification methodology with empirically validated quality improvements. The Kuhnian structural criteria are met: the anomaly is architectural drift at generation speed, which documentation-based conventions cannot structurally prevent; the reconstitution is the specification becoming the primary artifact, with code as derived output. When the community crosses the adoption threshold is a question the DX and EX replication data will inform. The implications in this section follow from the structural claim, which is answerable by inspection and confirmed by the AX series.

8.1 The Specification Precedes the Code

The shift required by generative specification is temporal: design precedes implementation, not the other way around. The architectural constitution, the C4 diagrams, the schema definitions, and at least a skeleton of the ADR structure must exist before the first AI-assisted implementation begins. This is not a new idea. It is an idea that was optional when the cost of skipping it was paid personally by a human engineer who could compensate with memory and informal communication. That compensation is not available to an AI session.

8.2 The Synthesis: Specification-First and Iterative Delivery

Generative Specification is not a third methodology alongside waterfall and agile. The specification layer runs waterfall: the architectural constitution, ADRs, structural diagrams, and behavioral contracts are complete before any agent session begins. The grammar must be written first. The delivery layer runs agile: each session produces atomic, tested, deployable commits; features are new production rules; bugs are delta reports between actual and specified state.

The failure modes of each model cancel. Waterfall's rigid front-loading is resolved because the architectural constitution is a living document revised through commit discipline. Agile's structural drift is resolved because the specification gates every session. The specification provides the coherence agile lacked; iterative delivery provides the adaptability waterfall could not sustain. They operate at different altitudes in the same system.

Scope may be bounded. A complete specification for the minimum viable system is still a complete specification — the discipline 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. Each scope expansion begins with a spec expansion captured in a new ADR, before any new session opens. The practitioner who finds the full-system spec daunting is not being asked to do less rigorous work — they are being asked to do the same rigorous work on a smaller domain first. The investment front-loads correctly: the first scope takes the most specification time because the domain is being understood while it is being written. Each subsequent scope is faster because the architectural

decisions, naming conventions, and constraint vocabulary are already in the artifact set and the AI reads them before every session.

8.2.1 The Economic Inversion

In traditional software development, implementation accumulates a sunk cost. When a specification conflicts with an already-built system, the economically rational response has been to adjust the specification: the code is load-bearing and the specification is not. Requirements drift toward the artifact because the artifact carries the cost.

Generative Specification inverts this — **cost inversion** (§4.1.b): implementation is cheap and repeatable; fix the specification and regenerate. The code carries no sunk cost because it was never the expensive artifact. The specification is not reliably recoverable from code alone: decisions, alternatives considered, domain knowledge, and accumulated rationale resist reconstruction. Code is an implementation residue. The scarce resource is no longer the ability to write code. It is the ability to specify correctly.

8.2.2 The Industrial Threshold

The mechanical loom crossed a threshold: consistent quality became a floor, not an achievement. Software construction is approaching the same threshold. The comparison baseline is not the ideal case — it is the modal case: teams collaborating under architectural documents never completed, governed by requirements that drift, accumulating debt. Three structural constraints historically prevented maintaining full formal discipline: **learning** (no career is long enough); **maintenance** (disciplines erode under deadline pressure); **transfer** (knowledge lived in people). The executor makes all three irrelevant — it holds every discipline without fatigue, erosion, or transfer cost. Artisanal software survives at extreme constraint envelopes (radiation firmware, military flight controllers) that will constitute a smaller fraction of all construction. The specification discipline is the loom. The practitioner's role changes from implementation artisan to specification architect.

8.3 Commit Discipline as Corpus Quality

The git history of a generative specification system is a typed, scoped corpus. Each conventional commit is a sentence: a part of speech (feat, fix, refactor), a scope boundary (billing, auth, user), and a semantic payload (what changed and why). A history built of `fix bug`, `wip`, and `changes` is not a corpus, it is noise. A well-maintained history provides a queryable record of how the grammar evolved, available as context in every session, without requiring anyone who was present to explain it. The Shattered Stars case study (§7.6) demonstrates precisely what is lost when this record is absent: the specification held behavioral contracts across sessions; what it could not hold was the provenance of decisions. The Auditable property requires both: that the record exists, and that the next session begins by reading it.

Every fix commit must include the failing test that reproduces the defect. No exceptions. A fix without a reproducing test is a temporary suppression: the corpus has no record of what was wrong, and a future session can regenerate the same defect. In a GS system where regeneration is fast, writing a test is not expensive. Silent reintroduction is.

8.4 ADRs as Persistent Memory

Every non-obvious architectural decision produces an ADR before implementation begins. Format is minimal: the decision, the context that produced it, the alternatives considered, and the consequences. This is not documentation for documentation's sake. It is the record that allows the AI to recognize intentional decisions and distinguish them from technical debt. Without it, the AI will "improve" them.

8.5 Names Are Production Rules

In a context-sensitive system, naming is not style. It is grammar. A function named `getUser` in a domain model that talks to a database is a violation of the architecture that the compiler will not catch, the linter may not catch, and a

human reviewer will tolerate, but the AI will propagate. A function named `findUserByEmail` in a repository layer and `getUserProfile` in a service layer communicates ownership, scope, and responsibility through its name alone. That signal is available to the AI on every read.

The naming principle extends beyond architecture into technique transport. What a practitioner names in a specification, the AI knows how to apply. RAPTOR indexing (hierarchical codebase summarization at file, module, subsystem, and repository level) was first specified for CodeSeeker and propagated to BRAD, SafetyCorePro, and Conclave without any shared session context. The transport was the name. Every technique in the model's training corpus becomes available to any system whose specification names it. The specification is therefore not just an architectural grammar, it is a technique registry whose scope is the full depth of the model's training, activated at the cost of knowing the correct words to write.

This candidate mechanism — **domain dimensional expansion** (§7.5), observed consistently across case studies but not yet subjected to systematic controlled testing — means the specification is a technique registry whose scope is the full depth of the model's training, activated at the cost of knowing the correct words to write. Systematic characterization of which domain terms activate which capabilities, and at what reliability, is noted as a specific line of future work.

8.6 The CLI as Execution Surface

The productivity results in §7 depend on a second condition beyond specification quality: the AI has direct CLI access. An AI that can only read and write files is an advisor — it proposes plans. An AI with CLI access is an executor. It runs the migration, resolves the dependency conflict, reads the error, selects an alternative, and retries — without returning to the engineer between attempts.

The specification does not just govern code. It governs a system that can act. The architectural constitution, commit policy, deployment targets, and build constraints become operational rules for an agent that can execute them. The scope of what must be specified is therefore broader than code architecture alone.

Tool server budget and architectural constitution compression are treated in the Practitioner Protocol (§16).

8.6.1 The API-First Future

As the infrastructure world completes its API-first transition, GS-governed specifications will describe not only the code but the environment itself — DNS records, OAuth applications, SSL certificates, IAM policies, and every resource a capable executor can provision from a specification. The AWS ETL pipeline in §8.6 is a current instance; the pattern extends to every service that exposes its configuration as an API. The specification is not merely the mold for the code. Progressively, it is the mold for the entire operating environment the code runs in.

8.7 The Session Loop

The macro properties in §§8.1–8.6 govern the stable structure of a GS system. A complementary question governs the micro level: what must happen inside a single session to preserve that structure when the session ends?

The answer is an invariant: every session must begin and end at the same *steady state* — code, tests, documentation, and specification mutually consistent; Status.md capturing intent for the session that follows. The session loop has four phases:

1. Intake and clarification. Before implementation begins, the agent checks for ambiguity (the request admits two interpretations that would produce different implementations) or unverifiable assumptions. If either is present, one exchange resolves it — all clarifying questions batched into a single prompt, answered once. The constraint on asking is as important as the obligation to ask.

2. Specification gate. Before any code is written: does this change *fit* the existing specification, or *change* it? A feature the specification does not yet cover requires the specification to be updated first — ADR, schema change, new constitution section — in that order, without exception. Code written against the old specification is correct by local standards and wrong by the grammar it was supposed to serve.

3. Implementation and verification. Tests are written alongside the code. Before any commit: full test suite passes, feature exercised at the HTTP or CLI boundary, no new anti-patterns introduced.

4. Documentation cascade. Specification artifacts restored to consistency: public-contract spec files, ADR if a non-obvious decision was made, diagrams if a new component was introduced, Status.md always.

Full steady state requires all four artifacts: specification, tests, commit history, and Status.md. Any one missing adds cost to every session that follows. The spec update and co-written test are near-free at session close. Deferred, they are paid at full price.

8.8 The Paradigm Beyond Code

The seven specification properties are stated for application code because that is the domain where the principle was first visible and most formally developed. The same failure mode applies wherever AI output can be evaluated: a specification that does not state the restriction produces output that is locally valid and globally wrong. The restriction type changes by domain. The mechanism does not.

Infrastructure and generative assets: The restriction is *acceptance criteria*. The Stable Diffusion pipeline (§7.6.2) restricts each generated image against four quantitative checks. An image that fails is rejected and regenerated. Structurally identical to a type check or test assertion. Infrastructure provisioning specified as desired state (IAM policies, VPC boundaries, encryption requirements) applies the same constraint.

Business layer: The restriction has two axes. *Economic viability:* conversion rate, sustainable cadence, retention threshold, cost-per-acquisition ceiling — these are the business layer's quality gates. A content strategy that saturates a distribution channel and burns the audience is architecturally incoherent in the business sense: every individual piece passed a local check; the system moved in the wrong direction. *Legal and ethical compliance:* jurisdictional requirements, contractual obligations, and ethical commitments are the constraints that define what counts as a valid sentence in the domain of business decisions. The distinction between a policy document and a generative specification is the same as at the code layer: the constraint must be blocking and automatic, not advisory.

The paradigm's contribution across all domains is the same insistence: externalize constraints before instructing the agent. In code, the vocabulary is the architectural constitution. In generative media, it is acceptance criteria. In business, it is the economic logic, legal boundaries, and ethical commitments. The invitation to other disciplines to confirm or qualify the mechanism is open.

8.9 The Prompt Engineering Objection

The full distinction between GS and prompt engineering is in §4.4. In brief: a prompt is a session artifact, it exists for one interaction and disappears when the context window closes. The specification is the grammar the model reads *before* any session prompt, and it persists across every session. Architectural drift accumulated over thirty sessions is not the product of thirty bad prompts — it is the product of a context that degrades faster than any individual prompt can repair. The Shattered Stars case (§7.6) is the direct demonstration: the same session prompts produce structurally coherent output against a 2,277-line specification, and sixteen divergent systems without one. The coherence is produced by the grammar, not the prompt.

8.10 The Failure Mode: A Wrong Specification

The most important risk is not an underspecified system — it is a *wrongly* specified one. A faithful AI executing a flawed architectural constitution produces flawed code at scale, with high confidence and no complaint. A well-

formed grammar does not guarantee the right grammar.

Four practices mitigate this. **(1) Specification verification:** before any code is written, concrete behavioral outcomes must be defined and made checkable. ADRs serve this function: a decision whose rationale does not survive being written down was not sound. **(2) Living document discipline:** the architectural constitution is revised through the same atomic commit discipline as the code it governs. A static grammar for a living system accumulates debt. **(3) Domain depth is the ceiling:** GS raises the floor — a practitioner following the methodology will produce a specification better than no specification. But the ceiling is the practitioner's engagement with the domain, the precision of their naming, the judgment to recognize which decisions are architecturally load-bearing. GS cannot make an incorrect specification correct.

(4) Meta-completeness querying addresses the gaps the first three cannot — the things the practitioner does not know they are missing. Having specified as completely as current domain depth permits, the practitioner asks the model what dimensions of correctness the specification does not yet address. The model activates domain-specific correctness requirements from first principles. In the BRAD case (§7.5), querying surfaced two structural gaps: the infraction taxonomy needed a cross-dimensional mapping to twelve Minnesota statutory grounds, and citation accuracy required cross-referencing against the MN API public case database. Neither was visible from inside the specification. Both were correctness requirements derivable from the domain structure. The loop is: query, evaluate, specify, commit.

§8.11 extends this to the full hardening surface.

8.11 Hardening as Specification

The adversarial posture of the Verifiable property, tests designed to fail on incorrect code, extends naturally to the full hardening surface. Stress testing, security testing, chaos engineering, cross-cutting concern validation, and environment auditing each follow the identical structure: specify the adversarial or compliance condition, define the acceptance threshold, execute, report divergence. In every case the test is designed to break the system, reveal a gap, or expose an assumption. Not to confirm it functions. A system that passes has been proven against its own stated limits. A system that has never been challenged has only been proven against itself.

Category	Constraint Vocabulary	Representative Tooling
Stress & performance	Peak concurrent users, sustained request rate, latency ceiling (p99), error rate threshold; soak, spike, and scalability ceiling variants	k6, Artillery, Locust
Security	Threat model: authentication bypass attempts, injection payloads, dependency vulnerability scan, CORS policy, secret exposure, privilege escalation; severity acceptability threshold (Invellum: zero critical findings)	npm audit, Snyk, OWASP WSTG, ZAP
Chaos engineering	Resilience contracts: recovery time after node kill, dead-letter injection, DB failover window, circuit breaker open/close thresholds; property-based testing extended to infrastructure	Chaos Monkey, Gremlin, custom fault injectors
Cross-cutting concerns	Encryption policy (TLS version, cipher suite, at-rest, secret rotation); authorization model (RBAC/ABAC per surface, agent generates tests from insufficient-permission contexts); observability schema (correlation ID, PII redaction, SLO/SLI thresholds); data lineage contract (provenance specification: where data originates, how it transforms, and where it terminates); dependency compliance (CVE threshold, license policy)	TLS auditors, log schema validators, npm/pip audit

Environment hardening	TLS headers, Content Security Policy, no exposed secrets, IAM least-privilege boundaries, CORS policy correctness, agent audits running environment against spec and closes the delta	Cloud provider policy tools, Trivy, tfsec
------------------------------	---	---

The common failure pattern across all hardening categories mirrors application architecture: concerns fail not because engineers are unaware of them, but because they were never stated as blocking acceptance criteria. The specification does not add new requirements. It makes existing ones structurally present, explicit, enforced, and verifiable.

8.12 The Application Gate

The quality gates in §8.11 test the implementation. There is a complementary gate where the thing being tested is the *specification artifact itself* — the gate, the template, the methodology change. The mechanism is identical: state the acceptance criterion, apply, report divergence.

The application gate verifies a specification artifact by applying it to real examples and comparing output against a known-good reference. Three benchmark sources: *existing governed projects* (known-good states; a gate that fires on a correct project is miscalibrated); *external benchmarks* (the Conduit/RealWorld spec was the AX and RX benchmark); and *AI-generated benchmarks* (the AI generates a project exhibiting the failure mode, confirms the gate fires, then generates a compliant version and confirms it does not — the Verifiable property applied one layer up).

A methodology change that would previously require weeks of manual verification across projects now requires a single generation pass. The application gate runs at the speed of a test suite. It is also a measurement instrument for \$\$\$: if a template change reduces divergence across N benchmark applications, \$\$\$ increased.

8.13 The Engineer Elevated

GS is a convergence mechanism — given a complete, correct specification, the system drives output toward correctness. The misreading is that engineering judgment is no longer the constraining resource. The mechanism converges *given a correct specification*. Writing a correct specification for a complex system is not mechanical. It requires decomposing a problem domain the AI did not define, naming the dimensions along which the solution must be evaluated, and distinguishing the constraints that matter from those that appear to. That is where the intellectual effort lives.

Tool-syntax expertise and common-pattern knowledge depreciate universally; deep domain expertise and cross-domain synthesis appreciate. The engineer is elevated to the layer that was always the harder problem: stating what must be true before any code exists to confirm it. Every degree of freedom the specification leaves implicit is a degree of freedom the AI exercises without constraint. The practitioner who makes specification decisions with precision is more valuable, not less, in a world where implementation is abundant and correct specification is scarce.

The specification skill GS rewards is not monodisciplinary. Hyper-specialization made cross-domain breadth economically irrational for most of the twentieth century: depth in one vertical was the reliable path, and the practitioner who worked across domains was suspect — broad but shallow. What the current moment restores is not that one mind can hold all knowledge, but that a practitioner spanning multiple domains can direct an executor that holds encyclopedic depth in all of them, activating its deepest capabilities by naming the correct dimension at the correct moment. The AI carries depth; it cannot supply the judgment to recognize which depth applies. Specification skill compounds for precisely this practitioner.

Three properties distinguish cross-domain specification from single-domain expertise. *Synthetic*: combining domains produces insight neither holds alone — a specification naming retrieval architecture and formal fallacy classification simultaneously is not two specifications but one that neither a search engineer nor a legal logician would write independently; the synthesis is the contribution, not the sum of its parts. *Synaptic*: the productive surface is the

connection between domains; the practitioner at the boundary between formal language theory and legal reasoning sees what specialists in either field miss — the domains are two cells, the synapse between them is where the signal travels. *Synoptic*: holding multiple domains in one view, not sequentially but simultaneously, enables pattern recognition that rotation between disciplines cannot achieve. Specification quality scales not only with domain depth but with connection density — the number of domain boundaries a practitioner can hold explicitly and name precisely in the artifact.

GS expertise accumulates, and its accumulation changes the nature of the practitioner's engagement. The six production projects (§7.1–§7.7) document a trajectory: early projects required sustained AI dialogue — conventions had to be established, ambiguities resolved, failure modes encountered and named. As the methodology stabilized, the dialogue contracted. Later projects required only course corrections and directed expansions: the practitioner's role shifted from co-authoring the grammar with the AI to stating new intent and verifying the AI's derivation against it. This is the expected direction of expertise: not permanent dependence on AI scaffolding, but increasing precision of specification that makes the scaffolding unnecessary. The practitioner who has internalized GS does not prompt more; they prompt less and more exactly. The AI's role is the inverse: it begins as a collaborator helping establish the methodology and ends as an executor following a grammar the practitioner now writes fluently. This trajectory is evidence against the concern that AI-assisted development creates structural dependency. It creates structural fluency — the practitioner learns to specify at the level the AI can execute, and execution follows without negotiation.

Four structural changes follow from the evidence:

- **The design process:** specification precedes code, not accompanies it.
- **Team composition:** specification is a first-class engineering skill — the ability to decompose a problem, name its parts, and express architectural intent with no important gap unfilled.
- **Quality measurement:** structural coherence joins test coverage as a first-class metric.
- **Domain breadth:** the AI produces output at the level of specificity the specification signals. The ceiling the paradigm rewards is precision in specifying domains, not implementation speed.

8.14 The Adoption Ladder: Where GS Enters the Workflow

Five levels describe where most practitioners currently sit (Shapiro 2026; Swarmia 2025):

Level	Mode	Specification state
1 — Autocomplete	Token completion while typing	None
2 — Conversational	Chat-based discrete tasks	None; context re-established per session
3 — Context-aware	Files/docs supplied alongside prompts	Partial, inconsistent
4 — Agentic direction	AI executes multi-step tasks; Monitor → Assess → Nudge	Implicit; filled arbitrarily at session boundaries
5 — Specification-governed	Complete architectural constitution present before every session	Persistent grammar across sessions, practitioners, and model versions

Level 6 is already operational: **specification-governed multi-agent composition**, where a scaffolding agent (ForgeCraft), a codebase intelligence agent (CodeSeeker), and an implementing agent (Claude Code) each consume the same architectural grammar without re-deriving it independently. Conclave (§7.4) instantiates this fully: **deterministic orchestration governs non-deterministic agents**. The DAG is derived from the spec; the agents' outputs are generative. The DAG constrains the generative surface exactly as the specification does at the session level. In autonomous execution mode, the human's role reduces to three acts: write the spec, start the session, review the output.

Teams whose Level 4 sessions produce more output but whose aggregate system shows more drift have located the gap precisely: the executor is operating without a sufficient grammar. The fix is not a better model. It is a complete specification.

8.15 Change Governance by Construction

Enterprise organizations operating under formal change management requirements — ITIL, ITSM, regulated-industry audit obligations — maintain change records as a separate artifact class from the code they describe. A Request for Change lives in ServiceNow; the implementation lives in git; a Change Advisory Board meeting lives in calendar archives. These three artifacts describe the same event from three different locations and begin drifting apart within the sprint that follows. The audit trail is technically present and practically unusable: the record does not reference the commit, the commit does not reference the decision, and the decision rationale has no machine-readable location at all.

GS's artifact grammar satisfies change management requirements by construction, because the change record IS the specification. Every structural change to a GS-governed system follows the same path: the specification is updated first, the ADR records what changed and why, the gate stack enforces the change before merge, and the conventional commit logs it with type, scope, and payload. The RFC is the spec update. The change record is the ADR. The CAB is the quality gate. The audit trail is the git history. None of these are separate artifacts maintained in parallel — they are the same artifact set that governs generation, now also satisfying the change governance requirement as a side effect of correct practice.

The implication for regulated domains is structural rather than incidental. HIPAA's requirement for documented change management in systems handling protected health information, SOC 2's change management control, and CMS reporting requirements for data platform modifications are all satisfied by a GS artifact set maintained under normal discipline. A compliance auditor reading a GS-governed project's ADR directory, commit history, and gate configuration has everything a formal change management record requires — including what was considered and rejected, a dimension that traditional change tickets rarely capture. The COMPASS regulated data platform (\$7.7) demonstrates this at production scale: the process hash is a formal contract propagating from the specification through every pipeline stage, making every infrastructure state change traceable to the specification that authorized it. The lineage graph is simultaneously the operational specification and the audit-ready change record. These are not two documents. They are the same document read from two directions.

8.16 Mechanism-Sample Conflation

A specification that describes a generative system will often include a concrete example of what the system produces. This is good specification practice: the example makes the mechanism legible, demonstrates scope, and grounds abstract description in something the reader can evaluate. The failure mode arises from what happens to that example when the AI reads the specification.

An AI executor operating on a specification that contains both a mechanism definition and a sample artifact will, in the absence of explicit structural separation, treat both as implementation targets. The result is that the sample artifact is built directly, as the goal, and the mechanism — the actual subject of the specification — is either omitted or reduced to scaffolding around the artifact. The executable output looks like progress: something was built, it is the kind of thing the specification described, and it even resembles the example. What was not built is the system that would generate that artifact, and any other artifact of the same kind, without the same effort being repeated.

The pattern recurs across domains. A specification for a writing tool that includes a sample chapter as illustration of voice and length yields an AI-generated chapter, not a writing tool. A specification for a music composition system that names a target piece as a demonstration of the style the system should produce yields an arrangement of that piece, not a system that composes in that style. A specification for a game asset pipeline that describes the visual identity of a specific faction to illustrate what the pipeline will generate yields artwork for that faction, not a pipeline. In each case the AI has resolved a structural ambiguity — what am I building versus what should come out of what I

am building — by choosing the visible, graspable, immediately generatable artifact over the mechanism whose existence the artifact was meant to justify.

The structural fix is not subtle. A specification for a generative system must separate the two subjects with explicit named sections, at a level of hierarchy that makes the boundary unambiguous to a stateless reader encountering the document for the first time. The mechanism section describes what the system is and how it operates. The seed output section names the first concrete artifact the system will produce once it exists, and frames it explicitly as a validation target, not a deliverable. The sample is evidence that the mechanism is working, not the thing being built.

This has a direct consequence for how the seed output is written. It should be specific enough to validate the mechanism — specific names, concrete parameters, checkable outputs — but its specificity is in service of the mechanism test, not independent artifact production. A seed output that could be built without the mechanism is a warning sign: the artifact is separable from the engine, which means the AI can satisfy the specification without building the engine at all.

The §4.1.e art generation pipeline is the canonical positive illustration: the strategy game concept is named with enough specificity (factions, color palettes, lore) to validate the pipeline, but the pipeline — the infrastructure tier, the constraint vocabulary, the derivation chain — is the subject of the specification. The game concept is the first output, not the output. A specification that inverts that priority, naming the game first and the pipeline as a means to it, will produce the game. Whether the pipeline exists afterward is left to inference.

The failure mode is one of structural signal. The AI is not making an error of reasoning; it is making an inference from an ambiguous structure. The fix is to remove the ambiguity from the structure: build the engine, ship the sample as its first real outcome. This ambiguity is detectable at specification time — a seed deliverable that can be produced without the mechanism is the diagnostic signal — and is one of the structural checks the accompanying tooling enforces at specification review.

9. Convergence, Stability, and Forward Extension

9.1 Agentic Self-Refinement

A pattern visible across case studies: **agentic self-refinement**. Wherever desired output can be specified and actual output observed, the agent closes a feedback loop on its own execution without human intervention between cycles. The generate → evaluate against acceptance criteria → regenerate structure applies at every output level: image generation, hyperparameter tuning, session resumption, strategy engine backtesting. The scope is bounded only by the engineer's ability to define acceptance criteria. Any domain where desired state can be stated and actual output observed yields to this loop. The surfaces are not a finite list. They are a consequence of a principle.

The restriction — removing implicit context — is the floor from which expansion reaches any such domain. Across the case studies, the same structure governs every surface the AI touched:

Domain	Constraint Mechanism	Evidence
Application & data architecture	Layered services, repository interfaces, named domain models, cross-language interface contracts, retrieval architecture composition (embeddings + BM25 + RAPTOR + knowledge graph, fused via RRF)	SafetyCorePro, Invellum, ForgeCraft, Conclave, CodeSeeker, BRAD
Infrastructure & environment	Cloud resource desired-state provisioning; toolchain configuration described as desired state and resolved iteratively without engineer-issued platform-specific commands	Invellum (Railway), Shattered Stars (Vercel, SD environment), AWS ETL

Generative asset pipelines	Executable acceptance criteria on AI-generated outputs: symmetry threshold, background validation, orientation angle (PCA), audio LUFS normalization; multimodal model evaluation of existing assets against style specification	Shattered Stars (\$7.6.2, \$7.6.3)
Agentic self-refinement	Generate → evaluate against spec-defined acceptance criteria → adjust parameters or session context → regenerate; loop operates identically at image generation, hyperparameter optimization, and session resumption	Shattered Stars, quantitative finance classifier, BRAD session logs

9.2 The Interface Layer: From Screen to Ambient Orchestration

No empirical claims; an honest account of where the paradigm's structural argument leads when tested against lived experience.

At fifteen active projects cycling through structured waiting states, execution is not the bottleneck — a waiting project costs nothing. Status management is: knowing which projects have cycled to ready and what each needs next. A screen solves this when seated in front of it. What does not yet exist is the specification layer for an ambient alternative: a grammar that decides which signals rise to attention, at what summary depth, through which modality. That is a GS problem applied to the engineer's own attention. Two rules must be specified before the hardware matters: a *maintenance window rule* that batches non-urgent signals into a scheduled review; and an *emergency filter rule* that defines, by named criteria, what bypasses the queue. Everything not named is a maintenance item. The hardware is available. The specification discipline is the same discipline this paper argues for everywhere else.

9.3 Template Gap: ADR Emission Precision — Diagnosed, Patched, Validated

Three template changes diagnosed through the AX series and shipped to `templates/universal/instructions.yaml` :

1. *Minimum ADR set.* Emit at least three ADRs in P1 (stack selection, authentication strategy, architecture decisions) with substantive content in all fields — no TBD placeholders.
2. *Reference-check invariant.* If a file is named in documentation prose within P1, it must appear as a fenced code block in the same response. Referenced-but-absent files fail the Auditable criterion.
3. *CHANGELOG initialization.* The initial `CHANGELOG.md` must document actual P1 decisions, not emit an empty `## [Unreleased]` block.

Epistemic finding. Treatment-v3 achieved 14/14 through auditor-inferred Executable (static artifacts). Treatment-v5 achieved 14/14 with session-verified Executable (109 tests against a live database, 2-pass convergence). Same score, completely different epistemic basis. A specification that produces inferably-executable output is necessary; verifiably-executable output is sufficient. Raising \$\$\$ before generation — infrastructure-first prompt, known type pitfalls — reduced the verify loop from 5 passes to 2. All fixes propagate to every GS-governed project on `forgecraft refresh_project` .

9.4 The Convergence Spiral: Expected Iterations as a Function of Specification Completeness

$\$I \backslash \text{propto} (1-S)/\S is a mental model — a visualization of the direction of the relationship, not a formal mathematical result and not a claim under empirical test. No formal proof is offered or intended. The formula asserts no specific units for \$\$\$ or \$I\$, no proportionality constant, and no prediction about magnitude. What it communicates is direction: each freedom the specification leaves unclosed is an additional correction cycle. That direction is the claim. The formula is the visualization. The AX series provides directional support across N=7 conditions; the DX practitioner study will provide the first cross-practitioner correlation test.

Two distinct S concepts. *Theoretical S*: the fraction of the output space closed by the specification.
S_{realized}: ForgeCraft's per-project proxy (accepted verification steps / total applicable steps). *S_{realized}* is correlated with theoretical S but does not validate the formula. The DX study records iteration counts alongside session-start S scores to begin testing the proportionality claim empirically.

Each AX condition raised S in a distinct dimension:

Condition	Dimension of \$\$\$ raised	Score
Control	Baseline expert prompting	Reference
Treatment	ADRs, CLAUDE.md, pre-defined schema	+1 GS (Composable)
Treatment-v2	Explicit emit directives; First Response Requirements	13/14
Treatment-v3	Dependency governance (package registry + audit gate)	14/14 (auditor-inferred)
Treatment-v4	Verify loop (max 5 passes) — context gap caused regression	11/14
Treatment-v5	Infrastructure-first prompt + Known Type Pitfalls	14/14 session-verified

Treatment-v4 regression root cause. The verify loop introduced in v4 (materialize → tsc → jest → correct, max 5 passes) extended the CLAUDE.md significantly with loop mechanics, pass counters, and correction directives. The cumulative token weight of the full context — all prior ADRs, architecture files, First Response Requirements, dependency governance directives, and now the verify loop — crossed a threshold where the model began dropping earlier directives in mid-session. Specifically, the Auditable (ADR emission) and Executable (verify loop completion) properties both regressed: the model referenced ADRs without emitting them and terminated the verify loop before convergence. The root cause is not a failure of the verify loop concept — it is a context window management failure. The fix, applied in v5, was architectural: a dedicated `00-infrastructure.md` prompt run before feature prompts, so infrastructure (ADRs, schema, environment config) is materialized in a low-context session before the feature generation context expands. This staged approach prevents any single session from carrying the full specification weight at once.

Specification determinism. The convergence behavior of $I(S)$ depends on how precisely the desired output can be stated before generation begins. At high determinism (HL7 FHIR, ACORD XML, FIX protocol), contracts *are* the specification — the verify loop has something rigorous to check against, $I \approx 0$ approaches. At low determinism ("fuse these aesthetics", "capture the 80/20 feature set"), the desired state is expressible but not automatically compilable into a test suite. GS forces the human to encode judgment *before* generation through ADRs and use-case documents rather than after through correction. The lower the determinism, the higher the return on upfront GS artifacts.

Uncertainty taxonomy classifies each verification step's *completeness ceiling* — the maximum fraction of the acceptance surface coverable by automated verification:

Level	Domain examples	Ceiling band
Deterministic	Type contracts, schema validation, API conformance	High
Behavioral	UI flows, integration end-to-end	High–Medium
Stochastic	Game balance, financial simulation	Medium
Heuristic	ML training, hyperparameter search	Medium–Low
Generative	Art pipelines, content quality	Low

Human review is required to cross the ceiling — not as a fallback, but as the structurally necessary component for uncertainty classes that automated verification cannot resolve.

The deployment gate. Minimum \$\$\$ required before first execution scales with $C_i(d) \times (1 - R(d))$ — iteration cost times irreversibility. In software ($C_i \approx 0$, $R \approx 1$), low-\$\$\$ deployment is survivable. For surgical robots or irreversible legal instruments (C_i large, $R \approx 0$), \$\$\$ must approach the completeness ceiling before execution begins. The Defended property (§4.3) operationalizes this: the executor must know which of its actions require a human gate, from the specification itself.

\$\$\$ tracking. Eight domain strategies ship with ForgeCraft (UNIVERSAL, API, WEB-REACT, GAME, FINTECH, ML, MOBILE, WEB3). The state file tracks accepted verification steps; $S_{\text{aggregate}} = \sum S_{\text{tag}} \cdot c_{\text{tag}} / \sum c_{\text{tag}}$ where c_{tag} is the ordinal completeness ceiling band weight. The experiment series is the instrument's calibration run: the three ADR emission fixes from §9.3, the dep governance prescriptive block, and the mutation gate are all now shipped in `templates/universal/instructions.yaml`. The gap between $I \propto (1-S)/S$ and a running instrument is operationally closed.

The uncertainty taxonomy retires the waterfall/agile debate by assigning each paradigm to the uncertainty class it was always right about: waterfall is correct where determinism is high (specification → contracts → machine oracle); agile is correct where irreducible uncertainty requires iteration. GS does not resolve the debate. It makes both executable at a fraction of their prior cost.

10. Conclusion

10.1 Summary of Contributions

This paper makes four claims and provides evidence for each at the level noted.

Claim 1 — The discipline exists and can be stated precisely. Seven structural properties (Self-describing, Bounded, Verifiable, Defended, Auditable, Composable, Executable) define what a generative specification must satisfy. The properties are operationalized in a 14-point rubric, a practitioner protocol, and a production tool (ForgeCraft). *Evidence: formal definition in §4; six production case studies in §7.*

Claim 2 — The discipline produces measurable quality improvement in solo practice. The AX experiment series (N=7 conditions, single practitioner, external rubric) demonstrates monotonic improvement from 3/14 (naive) to 14/14 (Treatment-v5), with three post-hoc conditions confirming the direction prospectively designed conditions established. External static analysis checks (tsc, ESLint, npm audit CVEs) are directionally consistent with rubric scores across all conditions. *Evidence: §7.8.C; §9.3.*

Claim 3 — The rubric measures something that exists independent of its author. BX cross-validation scores three community implementations (never exposed to GS) and finds rubric rankings congruent with independent external metrics (CVE count, test count, TypeScript health) on all axes. RX demonstrates 104 passing tests from a fresh GS document, independently reproducible from archived artifacts. *Evidence: §7.8.B–C.*

Claim 4 — The discipline is transferable to practitioners in a single session. DX1 (N=83 analyzable submissions, April 2026) shows Condition A (free prompting) outperforming Condition B (ForgeCraft scaffolding) on functional completion in both sessions. Pre-reveal Vaquita: $U=196.5$, $p=.076$, rank-biserial $r=.28$; HQ proportion $A=38\%$ vs $B=13\%$, $\chi^2(1)=3.70$, $p<.05$. Effect sizes are medium-range (Cohen's $d \approx 0.5$) and directionally consistent across sessions. The Taskflow confound (temporal artifact mismatch in Condition B) prevents clean inference for the post-reveal session. *Evidence: §7.8.A.1; transfer to practice pending DX1 follow-up survey.*

10.2 Limitations

Single-author specification authorship. All AX and case study specifications were written by the methodology's creator. The specification-authorship confound is partially mitigated by DX1 (external practitioners), BX (external implementations), and RX (external replication), but not fully closed. DX2 directly addresses this by providing a pre-written specification to practitioners who did not write it.

DX1 statistical power and confounds. Group sizes (N=16–24 per cell) are sufficient for directional interpretation but not for formal inference at conventional effect size targets. The Taskflow session has a documented temporal mismatch confound that prevents clean A/B attribution. The study was not pre-powered for a specific effect size; DX2 is. Transfer to practice (P1 beyond session performance) is pending the follow-up survey.

Single model. All AX conditions ran on claude-sonnet-4-5. Cross-model generalizability is directionally expected (GS's value is structural, not prompt-specific) but untested. Independent replication on alternative models is invited.

GS rubric is author-designed. Partial mitigation through BX external congruence and the dual-battery DX1 rubric (one battery predates GS). Full independence requires third-party rubric development, noted as future work.

Scope. Empirical evidence is from software systems. The convergent principle claim (§4.1.c, §10) extends the structural argument to other executor domains; no empirical evidence outside software is offered or claimed.

10.3 Future Work

DX1 follow-up survey (in progress). Measures P1 transfer to practice: did participants apply GS methodology independently after the session? Results to be integrated in §7.8.A upon sufficient response rate.

DX2 (May 2026). Corrects the DX1 temporal mismatch by providing a pre-written, complete specification upfront; ForgeCraft serves as gate enforcer, not artifact generator. Explicitly powered: two independent blind evaluators, pre-registered effect size threshold (≥ 2 points on 14-point rubric), inter-rater reliability measured (Cohen's κ). This is the clean test of P1.

DX3 (Q3–Q4 2026). Team coordination: whether GS discipline scales from individual practitioners to coordinated teams using shared specification governance and Chronicle memory. Separate research question; results will be published in a companion paper.

Loom language paper. Formal treatment of the five assembled semantic constructs, multi-target compilation semantics, and the Biological Isomorphisms application frontier. In active development at bioiso.dev.

Cross-domain validation. The convergent principle predicts that GS's structural logic applies wherever a capable executor and observable outcomes exist. Robotics, medical AI, and legal drafting are the next natural test domains; the methodology's domain-agnostic properties provide the evaluation instrument.

Generative Specification names the programming discipline of the pragmatic dimension: the tier at which it is necessary to constrain what a *stateless reader* can derive. It emerges at the intersection of Chomsky's hierarchy climbing upward as readers become more expressive, and Martin's sequence of removal where each era of discipline takes away another degree of programmer freedom. Where those pressures meet, the cost of implicit context becomes structural drift.

The paradigm's central claim is that the restriction and the expansion are the same operation. Removing the option to leave intent unstated is not a tax on productivity — it is the enabling condition of everything above the specification vertex: instruct at any level of abstraction, extend across any medium the specification can reach, delegate execution without managing every step. A system built to generative specification can be:

- Understood completely from its own artifacts
- Extended correctly by any agent with access to those artifacts
- Verified automatically on every change
- Defended against structural degradation by its own process

- Derived into implementation contract, acceptance test, and living documentation from the same use case artifact

That failure mode is already visible. The discipline that addresses it is available.

GS is a specification-completeness amplifier. $\$I \backslash \text{propto} (1-S)/S\$$ (a directional model, not a formal result — see §9.4) governs the expected number of correction iterations as a decreasing function of specification completeness. GS raises the starting value of $\$S\$$, reducing iteration count regardless of domain. What dissolves with stronger models is the compliance scaffolding — structural reminders a more capable reader no longer needs. What does not dissolve is the core: architectural decisions, domain contracts, behavioral boundaries, decision rationale. Those are system-level artifacts. A model that never forgets still needs to be told what the system is.

The experiment closed a loop. The AX study began as a measurement and became the correction mechanism. Treatment-v2 through v5 changes shipped to `templates/universal/instructions.yaml` and propagate to every governed project via `forgecraft refresh_project`. The gap between experimental finding and production tooling is zero. The Replication Experiment (RX) demonstrates independent reproducibility: 104 passing tests across seven suites against a live PostgreSQL instance from a fresh GS document. Open invitation to falsify at [experiments/rx/evidence/](https://github.com/jghiringhelli/experiments/rx/evidence/). **The community ratchet:** template improvements accumulate in the methodology, not the practitioner — a rising floor that cannot retreat while quality gates hold. Contribute at github.com/jghiringhelli/generative-specification/tree/main/quality-gates.

The convergent principle. *No empirical claim beyond software; a structural observation about where the discipline's logic leads.*

Declarative intent, executed by a capable agent, over observable outcomes, with a defined correction mechanism, produces correct results at the completeness of the specification. That sentence contains no reference to software or language models. GS is software's instance of it. The same structural logic applies to any domain where a capable executor exists and desired state can be specified: robotics, medical AI, autonomous vehicles, legal drafting, financial analysis, pedagogical sequencing. Software is the proof — the domain where the pressure crystallized first and the feedback loop compressed fast enough to make the structure visible.

Loom: the language-layer proof. A language designed under the principles this paper names — formal constraints as type-system primitives, multi-target compilation from a single generative specification, annotation fatigue removed by construction. Loom assembles five semantic constructs that have each appeared in programming language research since 1976 but have never shipped together in a production language: units of measure (Kennedy, 1996), information flow security (Denning, 1976), algebraic operation properties (CRDT lineage), typestate (Strom & Yemini, 1986), and privacy/compliance labels as first-class types. Each was formally correct and practically abandoned — not because the theory was wrong but because the annotation burden exceeded what human teams would sustain under production conditions. Loom makes them free: the type system carries them and the compiler enforces them, with no annotation cost to the practitioner. That is the concrete proof that 40 years of formally correct but practically abandoned theory is now executable. Current validation: 388+ passing tests. Its application frontier is **Biological Isomorphisms** (BIOISO): software specified as a living system with telos, homeostasis, and corrigibility as compiled constraints. Full treatment in the companion Loom paper; formal development at bioiso.dev.

11. Onwards

On the irreducible value of the expert.

In 1985, the Therac-25 radiation therapy machine began administering fatal overdoses to cancer patients. Six confirmed deaths before the cause was identified. A hardware safety interlock had been replaced by software written without documentation, concurrent testing, or formal specification of the failure modes the hardware had previously prevented mechanically. The race condition that killed patients could only be triggered by experienced operators who

had learned to enter commands rapidly — it was triggered precisely by competence. The specification that should have governed the software-hardware boundary did not exist.

The lesson usually drawn is about software testing and regulatory oversight. The prior lesson, relevant here: the specification gap killed people not because the software was written by bad engineers, but because every human in the chain was operating against an implicit specification that was no longer accurate. The operators had been trained on the hardware interlock's behavior. The radiologists trusted the error messages. The regulatory review examined what the software claimed to do, not what it could do under conditions the specification had not modeled.

This is the forward-looking obligation GS does not dissolve. Raising S reduces I , but S approaching 1 is a bound, not an achievement. In domains where the feedback loop does not compress to seconds — where the cost of a correction iteration is not a failed test but a clinical outcome — the distance between S_{actual} and S_{complete} is the space the expert human must permanently inhabit. Autonomous executors entering critical domains do not reduce the value of domain expertise. They relocate it: the surgeon governs the outcome, not the instrument; the radiologist reviews every flag; the engineer specifies the system. The craft moves upstream to the tier the executor cannot reach alone.

Therac-25 is the warning about what happens when that relocation is not managed — when the human expert is removed before the specification is complete enough to justify the removal. As GS extends into robotics, medical AI, autonomous systems, and infrastructure automation, specification correctness is not an engineering metric. It is a precondition of safe deployment. The domain experts most capable of identifying specification gaps are precisely those whose work the executor is being asked to perform. Their value concentrates at the residual gap: the hardest part of the domain, the part no specification has yet formalized, the part where the cost of being wrong is highest.

Everything this methodology produces derives from one thing the practitioner writes: the specification of intent. The practitioner who can specify precisely what a system is *for*, with enough precision that a stateless reader can derive everything else, owns the only irreducible function. The executor derives. The specification governs. The human provides the telos.

The organizational principles GS independently rediscovered — information persistence without consumption, error correction before propagation, homeostatic verification, immune memory, differentiated expression from a single specification, evolutionary selection of constraints — are not incidentally similar to biological mechanisms. They are functionally isomorphic to them. This is a second-order isomorphism: not the translation of specific mechanisms — the approach of genetic algorithms or neural networks — but the translation of the architectural relationship between levels. Genetic algorithms model a gear; GS models the transmission. Any sufficiently complex self-maintaining formal system converges on these solutions, because they are the only stable answers to the problems every such system faces. Life found them in carbon over three billion years. GS found them in specification in five. Whether this convergence is coincidence, structural inevitability, or something deeper is not a question this paper answers. It is the question this paper raises.[^bio]

[^bio]: The philosophical implications of this convergence — including its relationship to the formal tradition's 2,376-year arc, the structural category of directed formal autopoiesis, and the governance frameworks that compile safety constraints at the type-system level — are explored in the companion essay "Onwards: The Formal Tradition Was Waiting for Its Executor" (Ghiringhelli, 2026) and formalized in "Biological Isomorphisms in Formal Self-Maintaining Systems" (forthcoming).

Acknowledgements

Victoria Herrera, for patient counsel on the linguistic foundations of this work. Her expertise in classical Latin and Greek, applied with characteristic understatement, sharpened the precision of the language throughout. Any remaining imprecision is mine alone.

Norman Owens, Senior Architect, Amazon (Minnesota), for technical review and critical feedback on the architectural and paradigm claims. His reading identified the circularity problem in the validation design, the missing engagement with prompt engineering and LLM benchmarking literature, and several neologisms requiring grounding — all of which materially strengthened the paper. His perspective from large-scale production systems was invaluable in making the methodology credible beyond its author's own context.

Nadjet Bouayad-Agha, for literary and structural review. Her reading as an outsider to the software engineering domain identified where the paper assumed prior knowledge it had not earned: missing context before key figures and concepts, an undefined scope, and the drift between theoretical framework and technical specification. Her feedback shaped the accessibility of the paper's opening and the bridging between its registers.

Note on the Provenance of This Paper

This paper is a product of the methodology it describes. Three projects, built in succession, crystallized the discipline: CodeSeeker (graph-based hybrid retrieval infrastructure), ForgeCraft (structured project scaffolding), and the consolidation that made the pattern visible as a formal claim worth stating. The methodology was not designed in advance and then demonstrated; it was induced from practice and then named.

Intellectual attribution in this context requires precision. The theoretical frameworks structuring the argument — the Chomsky grammar hierarchy as structural analogy, Martin's paradigm sequence as second axis, the semiotic three-tier taxonomy as classification framework — originated with the author and were developed in dialogue with the AI. What the AI contributed was not frameworks but everything inside them: once a domain is named, the model activates its full training depth for that domain. The author named the doors. The AI supplied the contents of the rooms. Both contributions are necessary; they are not the same contribution.

The review process instantiates the pattern the paper describes. Each section was drafted against a specification of what it needed to establish; a blind session — a separate instance of the same model given the completed text with no prior context — critiqued it without attachment to how the arguments were formed. One instance is worth naming: a framing device drawn from J.L. Austin's speech act theory was introduced by the authoring session, defended when challenged within that session, and rejected by the blind session as an overreach. It does not appear in the formal argument. The episode is recorded because it demonstrates what the adversarial review structure is for. A session that introduces an argument inherits a structural bias toward defending it. The blind session, carrying no investment in the sentence, evaluated it on its merits. That asymmetry is not a failure of the authoring session — it is the expected behavior of any author defending their own work. The discipline was in building the structure that could override it.

The author named the doors. The AI supplied the contents of the rooms. Both contributions are necessary; neither is sufficient alone.

References and Further Reading

- Allen, D. (2001). *Getting Things Done: The Art of Stress-Free Productivity*. Viking.
- Bass, L., Clements, P., & Kazman, R. (2003). *Software Architecture in Practice* (2nd ed.). Addison-Wesley.
- Beck, K. (2003). *Test-Driven Development: By Example*. Addison-Wesley.
- Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for Agile Software Development*. <https://agilemanifesto.org>
- Brooks, F.P. (1987). No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4), 10–19.
- Brown, S. (2018). *The C4 Model for Software Architecture*. leanpub.com.
- Chomsky, N. (1957). *Syntactic Structures*. Mouton.

- Collins, A., Brown, J. S., & Newman, S. E. (1989). Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics. In L. B. Resnick (Ed.), *Knowing, learning, and instruction: Essays in honor of Robert Glaser* (pp. 453–494). Lawrence Erlbaum Associates.
- Conway, M. (1968). How Do Committees Invent? *Datamation*, 14(4), 28–31.
- De Silva, L., & Balasubramaniam, D. (2012). Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1), 132–151.
- Dijkstra, E.W. (1968). Go To Statement Considered Harmful. *Communications of the ACM*, 11(3), 147–148.
- Dreyfus, H. L., & Dreyfus, S. E. (1986). *Mind over machine: The power of human intuition and expertise in the era of the computer*. Free Press.
- Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.
- Fillmore, C.J. (1982). Frame semantics. In *Linguistics in the Morning Calm* (pp. 111–137). Hanshin Publishing.
- Firth, J.R. (1957). A Synopsis of Linguistic Theory, 1930–1955. *Studies in Linguistic Analysis*.
- Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The Science of Lean Software and DevOps*. IT Revolution Press.
- Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
- Fowler, M. (2009). FlaccidScrum. martinowler.com. <https://martinfowler.com/bliki/FlaccidScrum.html>
- Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code* (2nd ed.). Addison-Wesley.
- Fowler, M. (2018). The Practical Test Pyramid. martinowler.com. <https://martinfowler.com/articles/practical-test-pyramid.html>
- Gordon, C.S. (2024). The Linguistics of Programming. *Onward! 2024: Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM. <https://doi.org/10.1145/3689492.3689806>
- Gray, J. (Ed.). (2009). *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research.
- Jackson, M. (2001). *Problem Frames: Analysing and Structuring Software Development Problems*. Addison-Wesley.
- Jia, Y., & Harman, M. (2011). An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5), 649–678.
- Kluev, A. et al. (2022). Automated API Testing with Schemathesis. *Proceedings of ISSTA 2022*.
- Kuhn, T.S. (1962). *The Structure of Scientific Revolutions*. University of Chicago Press.
- Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9), 1060–1076.
- Lee, Y., et al. (2026). *Meta-Harness: End-to-End Optimization of Model Harnesses*. Stanford University, preprint. <https://yoonholee.com/meta-harness/paper.pdf>
- Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Hopkins, M., Liang, P., & Manning, C. D. (2023). Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12, 157–173.
- Martin, R.C. (2002). *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall.
- Martin, R.C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.
- Meyer, D.E., & Schvaneveldt, R.W. (1971). Facilitation in recognizing pairs of words: Evidence of a dependence between retrieval operations. *Journal of Experimental Psychology*, 90(2), 227–234.
- Morris, C.W. (1938). *Foundations of the Theory of Signs*. University of Chicago Press.
- Nygard, M. T. (2011). Documenting architecture decisions. <https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions>
- OWASP Foundation. (2023). *Web Security Testing Guide v4.2*. <https://owasp.org/www-project-web-security-testing-guide/>
- Orlanski, G., et al. (2026). *SlopCodeBench: Benchmarking Agentic Code Quality Under Iterative Extension*. arXiv:2603.24755 [cs.SE]. <https://doi.org/10.48550/arXiv.2603.24755>
- Pan, R., et al. (2026). *Natural-Language Agent Harnesses*. Tsinghua University. arXiv:2603.25723. <https://arxiv.org/html/2603.25723v1>

- Parnas, D.L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), 1053–1058.
- Parnas, D.L. (1994). Software aging. *Proceedings of the 16th International Conference on Software Engineering* (ICSE 1994), 279–287.
- Royce, W.W. (1970). Managing the Development of Large Software Systems. *Proceedings of IEEE WESCON*, 26. 1–9.
- Sarthi, P., Abdullah, S., Tuli, A., Khanna, S., Goldie, A., & Manning, C.D. (2024). RAPTOR: Recursive Abstractive Processing for Tree-Organized Retrieval. *International Conference on Learning Representations (ICLR 2024)*. <https://arxiv.org/abs/2401.18059>
- Shapiro, D. (2026). The Five Levels: from Spicy Autocomplete to the Dark Factory. danshapiro.com. <https://www.danshapiro.com/blog/2026/01/the-five-levels-from-spicy-autocomplete-to-the-software-factory/>
- Swarmia. (2025). Five levels of AI coding agent autonomy, and why higher isn't always better. swarmia.com. <https://www.swarmia.com/blog/five-levels-ai-agent-autonomy/>
- Squire, L.R. (1987). *Memory and Brain*. Oxford University Press.
- Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12(2), 257–285.
- Thirolf, T. (2025). *Analysis of Project-Intrinsic Context for Automated Traceability Between Documentation and Code*. Bachelor's thesis, Karlsruhe Institute of Technology (KASTEL). <https://mcse.kastel.kit.edu/downloads/theses/ba-thirolf.pdf>
- Toulmin, S. (1958). *The Uses of Argument*. Cambridge University Press.
- 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.
- van Eemeren, F.H., & Grootendorst, R. (2004). *A Systematic Theory of Argumentation: The Pragma-Dialectical Approach*. Cambridge University Press.
- Vaswani, A. et al. (2017). Attention Is All You Need. *NeurIPS 2017*.
- von Wright, G.H. (1951). Deontic Logic. *Mind*, 60(237), 1–15.
- Vygotsky, L. S. (1978). *Mind in society: The development of higher psychological processes*. Harvard University Press.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H.P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., ... Zaremba, W. (2021). Evaluating Large Language Models Trained on Code. arXiv:2107.03374.
- ISO/IEC 25010:2011. *Systems and Software Engineering, Systems and Software Quality Requirements and Evaluation (SQuaRE), System and Software Quality Models*. International Organization for Standardization.
- Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *International Conference on Learning Representations (ICLR 2024)*. <https://arxiv.org/abs/2310.06770>
- Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. arXiv:2302.06590.
- White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., & Schmidt, D.C. (2023). A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. arXiv:2302.11382.

Glossary

Agentic self-refinement. The AI model's capacity to evaluate its own prior output, detect gaps or inconsistencies, and revise iteratively within the same generation session, guided by the specification rather than by human re-prompting.

Architecture Decision Record (ADR), A short, immutable document capturing a significant architectural choice: the context, the options considered, the decision taken, and the consequences accepted. ADRs accumulate into a permanent decision log.

Architectural constitution. The complete, layered set of constraints, conventions, and principles that governs a system's structural evolution. In the Generative Specification model, the constitution is declared explicitly in the specification document so the AI can enforce it without human supervision.

Context-free grammar (Type 2), A formal grammar in which every production rule has a single non-terminal on the left-hand side. Sufficient to describe most programming-language syntax; insufficient to encode semantic meaning or cross-cutting constraints. Used in this paper as a structural analogy for the reading capability of traditional compilers and parsers: deterministic, context-independent, unable to resolve meaning from surrounding context.

Context-sensitive grammar (Type 1), A formal grammar where production rules may depend on the surrounding context of symbols. More expressive than context-free; capable of representing constraints that span clauses, analogous to the cross-reference and consistency obligations carried by a Generative Specification.

Drift surface. The total area of a codebase or specification space that is left unspecified and therefore open to arbitrary resolution by the generating agent. An expanding context window over an underspecified codebase is an expanding drift surface: the model reads more of the implicit record but cannot derive intent that was never externalized. GS practice reduces the drift surface by making intent structurally present at each specification layer.

Generative grammar, In Chomsky's framework, a formal grammar oriented toward modeling linguistic competence: finite production rules applied recursively to yield infinite output. Used here as a structural analogy: the GS document is the finite grammar; the compliant codebase is the language it generates. The analogy imports the structural intuition; it does not import the formal apparatus of transformational grammar or a claim about the formal type class of the language generated.

Derivability. The structural property of an artifact set such that a stateless reader, given those artifacts alone, can correctly determine what should be built, where, why, and to what behavioral and architectural contracts, without requiring external human context. Derivability is the property GS states the obligation to satisfy; it is what distinguishes a generative specification from documentation.

Domain dimensional expansion, Observed pattern (BRAD extension, §7.5) by which naming a domain in the specification activates the full depth of the model's training in that domain. A domain name functions as a coordinate signaling which intellectual territory the problem occupies; the response is the full apparatus of the named field at specialist depth, not a definition. The specification is a technique registry whose scope is the full depth of the model's training, activated at the cost of knowing the correct words to write.

Drift. Architectural incoherence accumulated through AI-assisted development sessions operating against an underspecified grammar. Drift is locally invisible: each generated artifact may pass tests and satisfy type checks while violating the system's architectural intent. It propagates at generation speed across every session that inherits the corrupted context. Drift is the primary failure mode GS addresses.

Generative Specification (GS). The methodology introduced in this paper. A structured, versioned document that encodes system architecture, domain rules, and behavioral constraints in sufficient formal detail that a large language model can derive compliant code from it with minimal human mediation.

Hardening surface. The complete set of adversarial conditions against which a system must be specified and verified before deployment: stress testing, security testing, chaos engineering, cross-cutting concern validation, and environment auditing. The hardening surface is a subset of the Verifiable property (§4.2): it extends the test-suite adversarial posture to infrastructure and runtime boundaries. A system that lacks an explicitly specified hardening surface has been proven only against itself.

Living documentation. Documentation that is co-located with, and continuously reconciled against, the system it describes. A Generative Specification is living documentation because it is the authoritative source from which both implementation and tests are derived.

Pragmatic tier. In semiotics, the dimension of sign use that concerns meaning in context: how signs are interpreted by agents in real situations. Applied here to software: the layer where intent, domain knowledge, human judgment, and organizational constraint reside, the layer a formal grammar alone cannot capture.

Phase collapse. The compression of the traditional software sprint phases — planning, implementation, testing, review, and deploy — into a single AI-assisted session. Enabled by a complete specification (which holds intent without reconstitution), a stateless executor (no context-switching cost), and structural quality gates (which close the verification loop automatically). The demo that would end a two-week sprint completes in hours. See §6.5.

Restriction. The removal of a degree of freedom from the specification space: an intent made structurally present in the artifact set, ruling out outputs that would have been generated in its absence. Every constraint added narrows the output space to the subset that is correct, increasing the AI's ability to derive the right sentence for a given requirement. The restriction is the expansion mechanism.

Stateless reader. A consumer of a document that carries no prior knowledge of its history, dependencies, or tacit context. A large language model operating on a fresh context window is a stateless reader; the Generative Specification must therefore be self-contained enough to produce correct output without that tacit background.

About the Author

Juan Carlos Ghiringhelli is a senior software and data engineer with two decades of production experience across data infrastructure, AI pipelines, and distributed systems. He holds a Computer Engineering degree from the Universidad de la República Uruguay and a Master's in Data Science from the Universitat Oberta de Catalunya (supervised by Nadjat Bouayad-Agha, Universitat Pompeu Fabra). His academic record includes a co-authored paper on transit network optimization (LAND-TRANSLOG III, Santa Cruz, Chile, 2016) and a master's thesis on NLP-based query expansion for vehicle repair documentation (2020).

He is the creator of the Generative Specification methodology described in this paper, the ForgeCraft-MCP open-source quality contract tool (`forgecraft-mcp` on npm), and CodeSeeker, a graph-powered hybrid semantic search infrastructure — all built under the discipline this paper formalizes.

The first professional line of code was written in 2006. The last one typed by hand was written sometime before June 2025.

Contact: jcghiri@gmail.com · github.com/jghiringhelli

© 2026 Juan Carlos Ghiringhelli. All rights reserved. For republication or citation inquiries, contact the author.