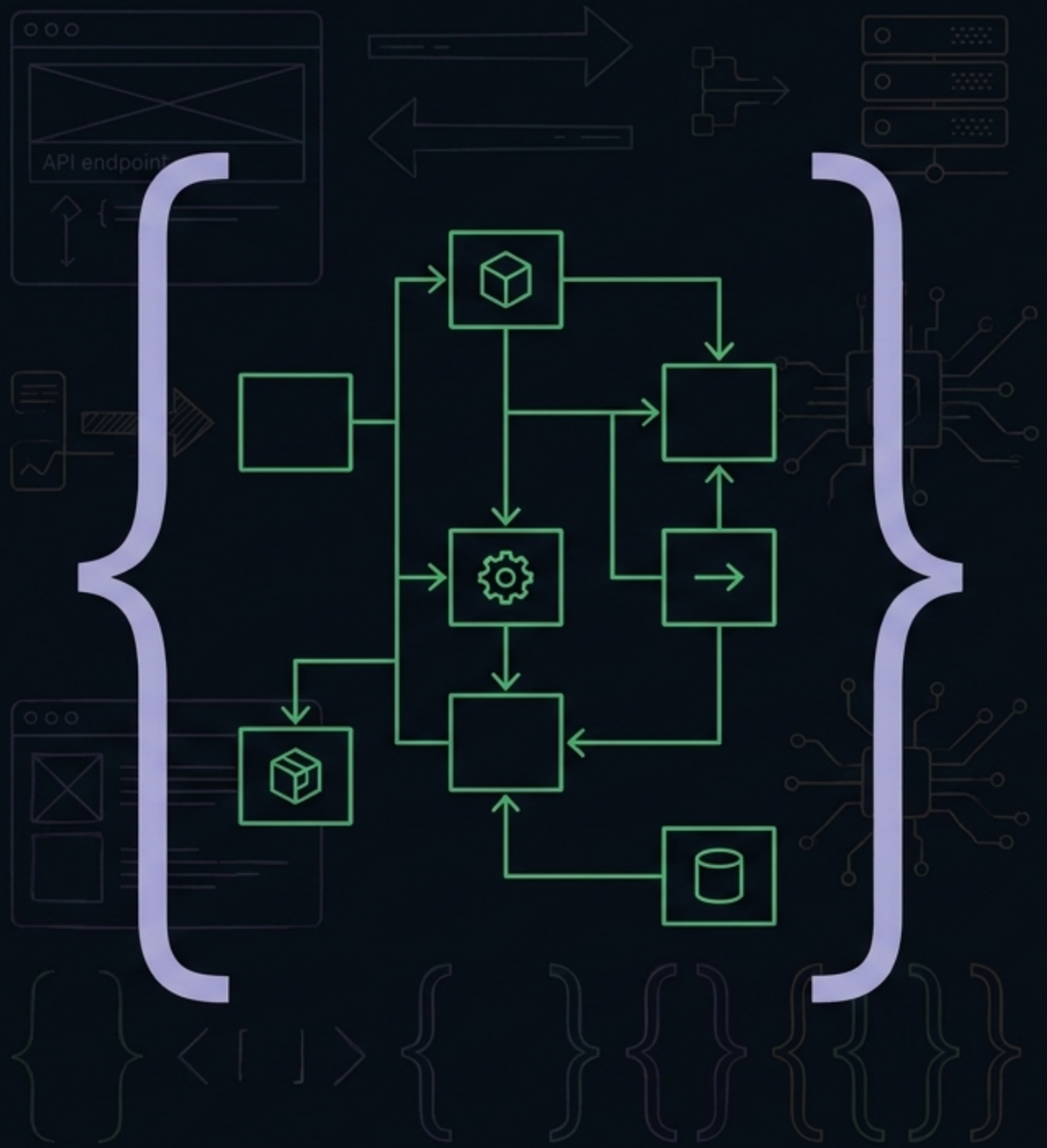


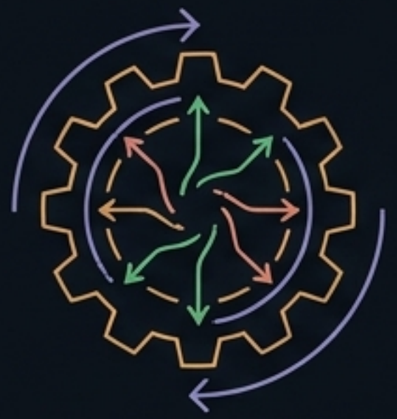
Enterprise Agentic Software Engineering

Mastering Instruction-Driven Development (IDD) and Deterministic AI Workflows

Version 3.0 / Architecture & Deployment Standards

The Illusion of Ad-Hoc Productivity

Inconsistency



The same prompt produces different results across sessions, models, or team members. Zero deterministic reliability.

Context Loss



AI forgets previous decisions. Each session starts from zero, causing architectural coherence to degrade over time.

No Collaboration



Prompts live exclusively in one developer's head. There is no shared understanding, review process, or handoff protocol.

Ad-hoc prompting treats AI interaction as a casual conversation.
Engineering requires a structured process.

The Ad-Hoc Spiral

Death by a Thousand Prompts



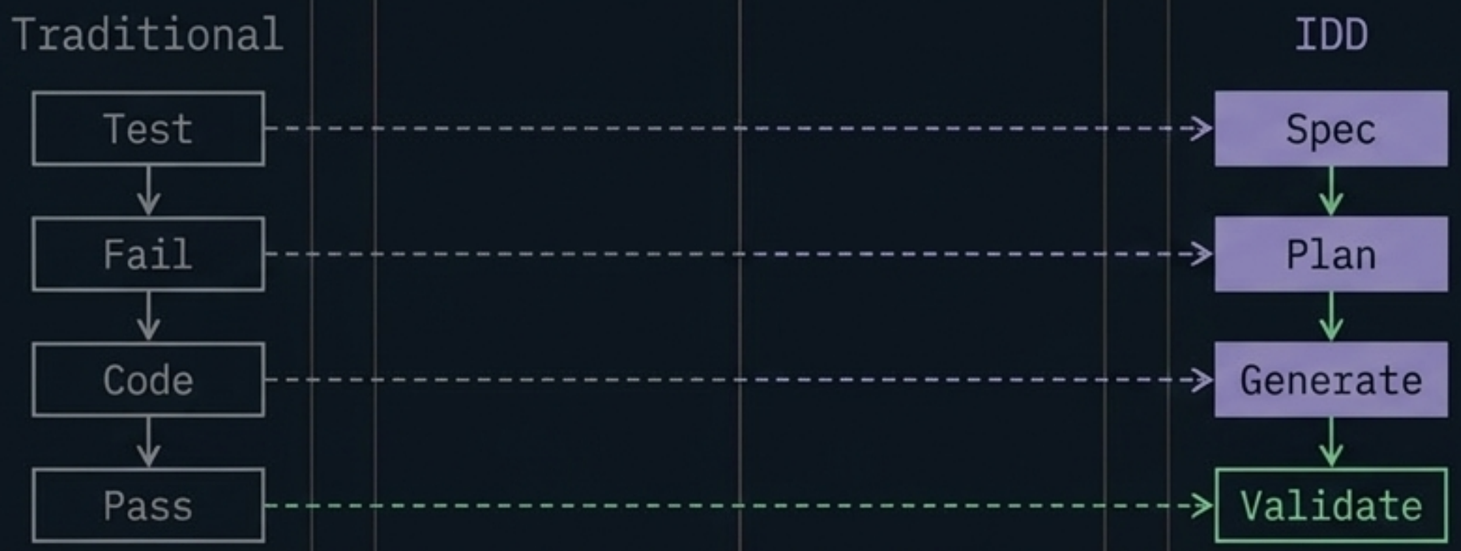
Without structure, each isolated AI interaction accumulates subtle inconsistencies.

The codebase becomes a patchwork of individually correct, but collectively incoherent components.

Core Anti-Pattern: Relying on ephemeral memory instead of persistent engineering contracts.

The Paradigm Shift: TDD for Prompts

Instruction-Driven Development (IDD) – also known as Specification-Driven Development (SDD) – is a methodology where detailed, structured instructions are written before engaging the AI.



Core Philosophy

Spec First: Clarify intent before requesting execution.

Artifacts: Specs are version-controlled and peer-reviewed.

Separation of Concerns: Humans own the what and why; AI executes the how.

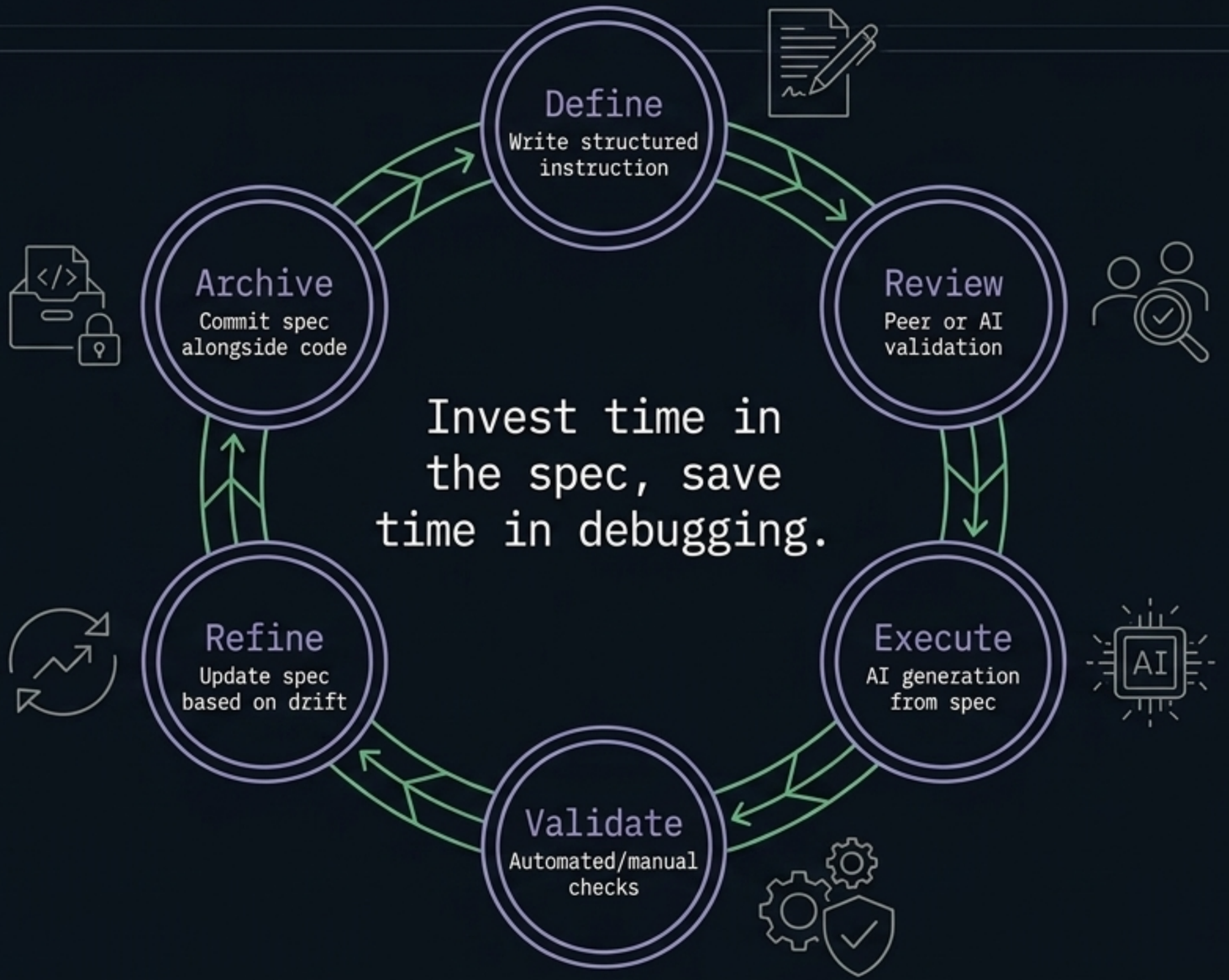
Deterministic Intent: Instructions form a repeatable contract.

The Evolution of Software Engineering

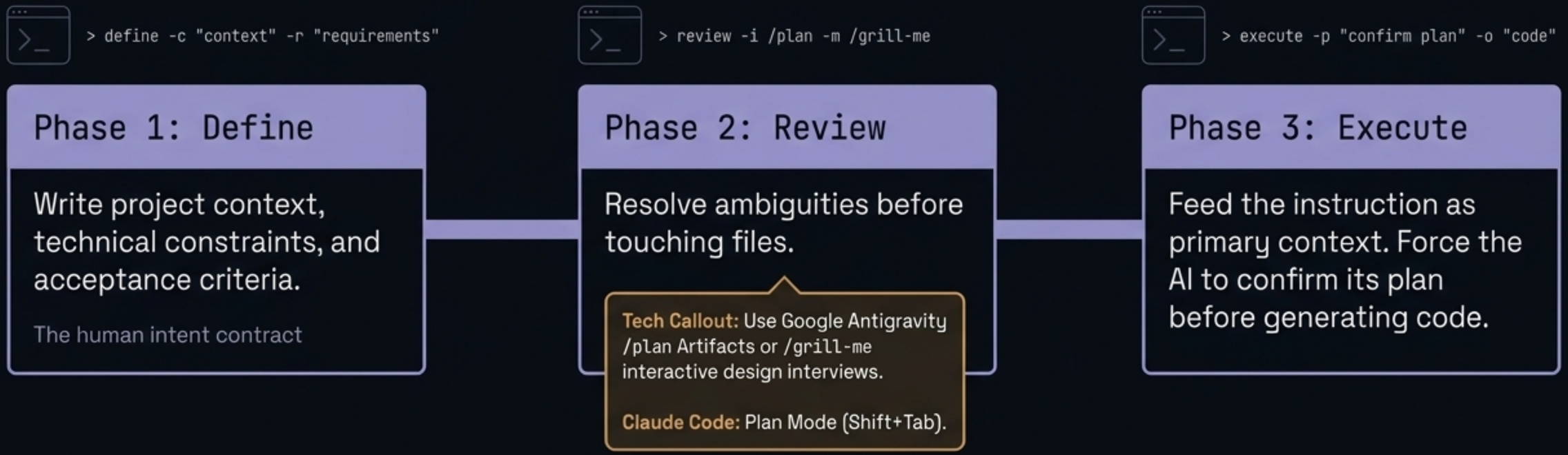
Dimension	Traditional Dev	Vibe Coding	IDD/SDD
Planning	Req docs / tickets	Mental model + ad-hoc	Structured spec doc + Plan Artifact
Reproducibility	High	Low / Fragile	High via spec + evalset
Scalability	High	Low / Tribal	High via shared specs & SKILL registries
Speed	Slow	Very fast but fragile	Fast after upfront spec investment
Quality Control	Code review + CI	Minimal	Spec review + hooks + eval CI

IDD doesn’t replace vibe coding — it matures it from exploration to production delivery.

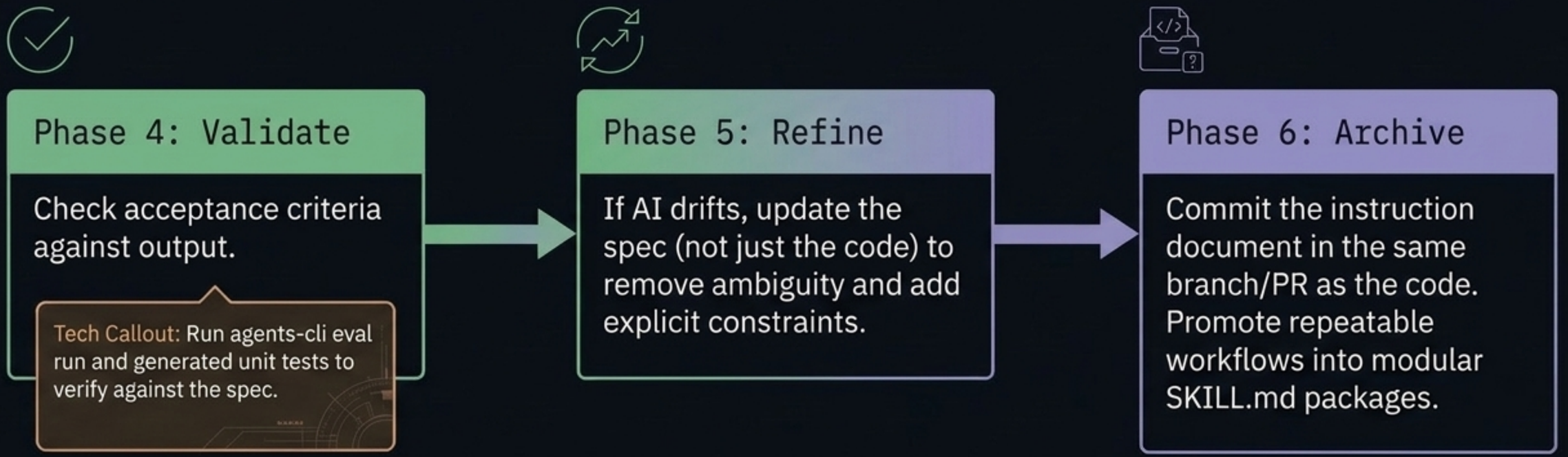
The 6-Phase Specification Pipeline



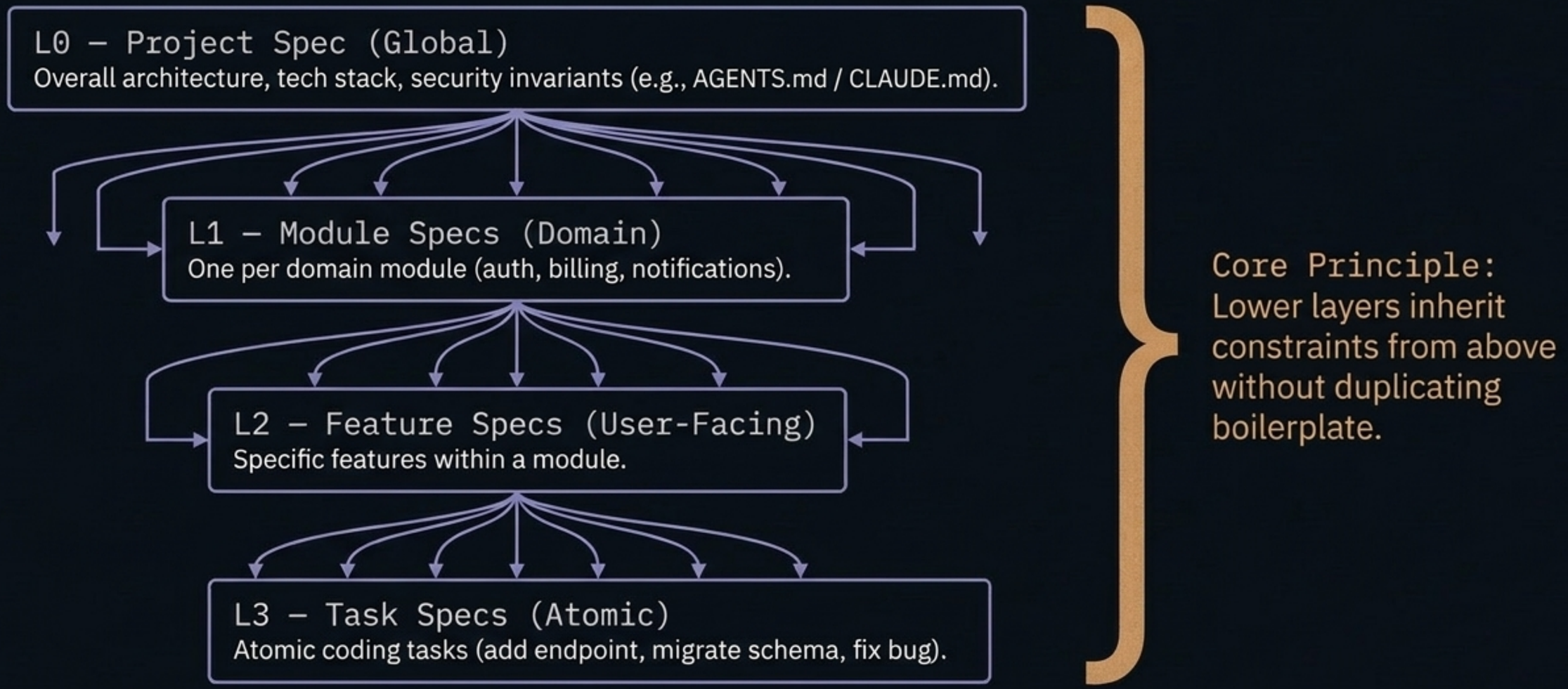
Phases 1—3: Human Alignment & Execution



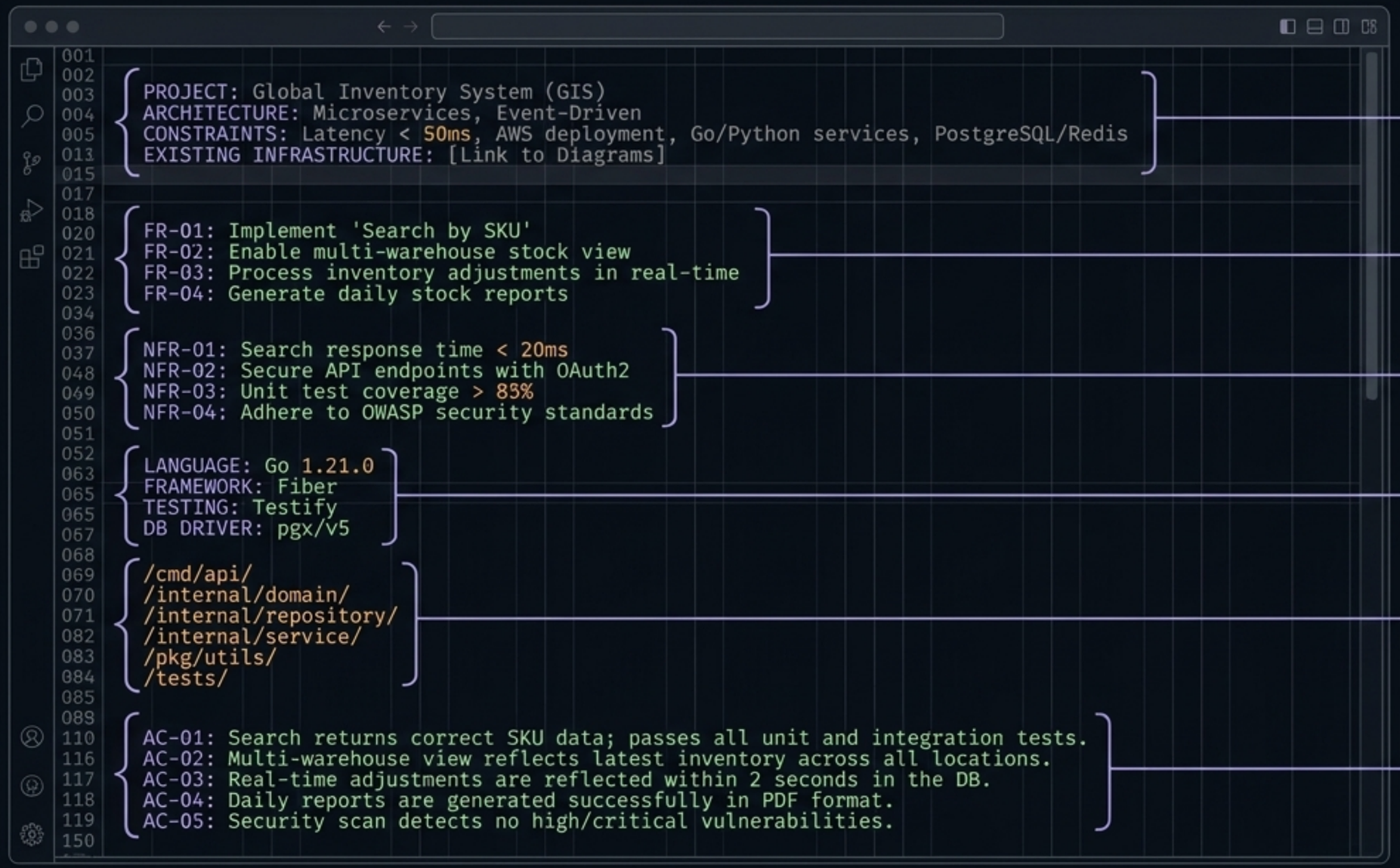
Phases 4–6: Verification & Persistence



Layered Specifications Architecture



Anatomy of an Instruction Document



```
PROJECT: Global Inventory System (GIS)
ARCHITECTURE: Microservices, Event-Driven
CONSTRAINTS: Latency < 50ms, AWS deployment, Go/Python services, PostgreSQL/Redis
EXISTING INFRASTRUCTURE: [Link to Diagrams]

FR-01: Implement 'Search by SKU'
FR-02: Enable multi-warehouse stock view
FR-03: Process inventory adjustments in real-time
FR-04: Generate daily stock reports

NFR-01: Search response time < 20ms
NFR-02: Secure API endpoints with OAuth2
NFR-03: Unit test coverage > 85%
NFR-04: Adhere to OWASP security standards

LANGUAGE: Go 1.21.0
FRAMEWORK: Fiber
TESTING: Testify
DB DRIVER: pgx/v5

/cmd/api/
/internal/domain/
/internal/repository/
/internal/service/
/pkg/utils/
/tests/

AC-01: Search returns correct SKU data; passes all unit and integration tests.
AC-02: Multi-warehouse view reflects latest inventory across all locations.
AC-03: Real-time adjustments are reflected within 2 seconds in the DB.
AC-04: Daily reports are generated successfully in PDF format.
AC-05: Security scan detects no high/critical vulnerabilities.
```

Context Block: Project awareness, existing architecture, inherited constraints.

Functional Requirements: Numbered, independently testable capabilities.

Non-Functional Requirements: Latency, security, and quality guardrails.

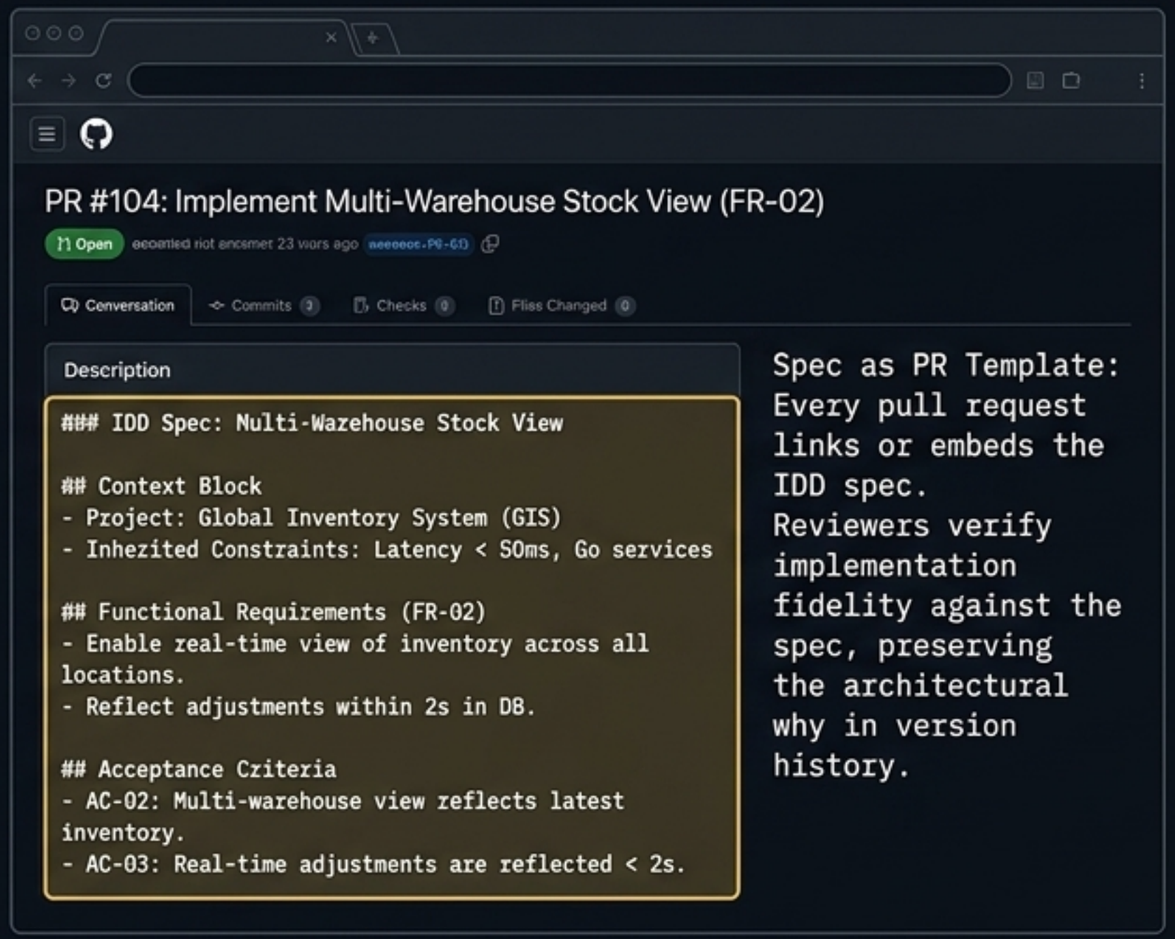
Technical Constraints: Locked-in language versions, frameworks, testing libraries.

File Structure: Explicit directory layout preventing ad-hoc file sprawl.

Acceptance Criteria: Verifiable pass/fail conditions for done.

The Spec-Test-Code Loop & Team Scalability

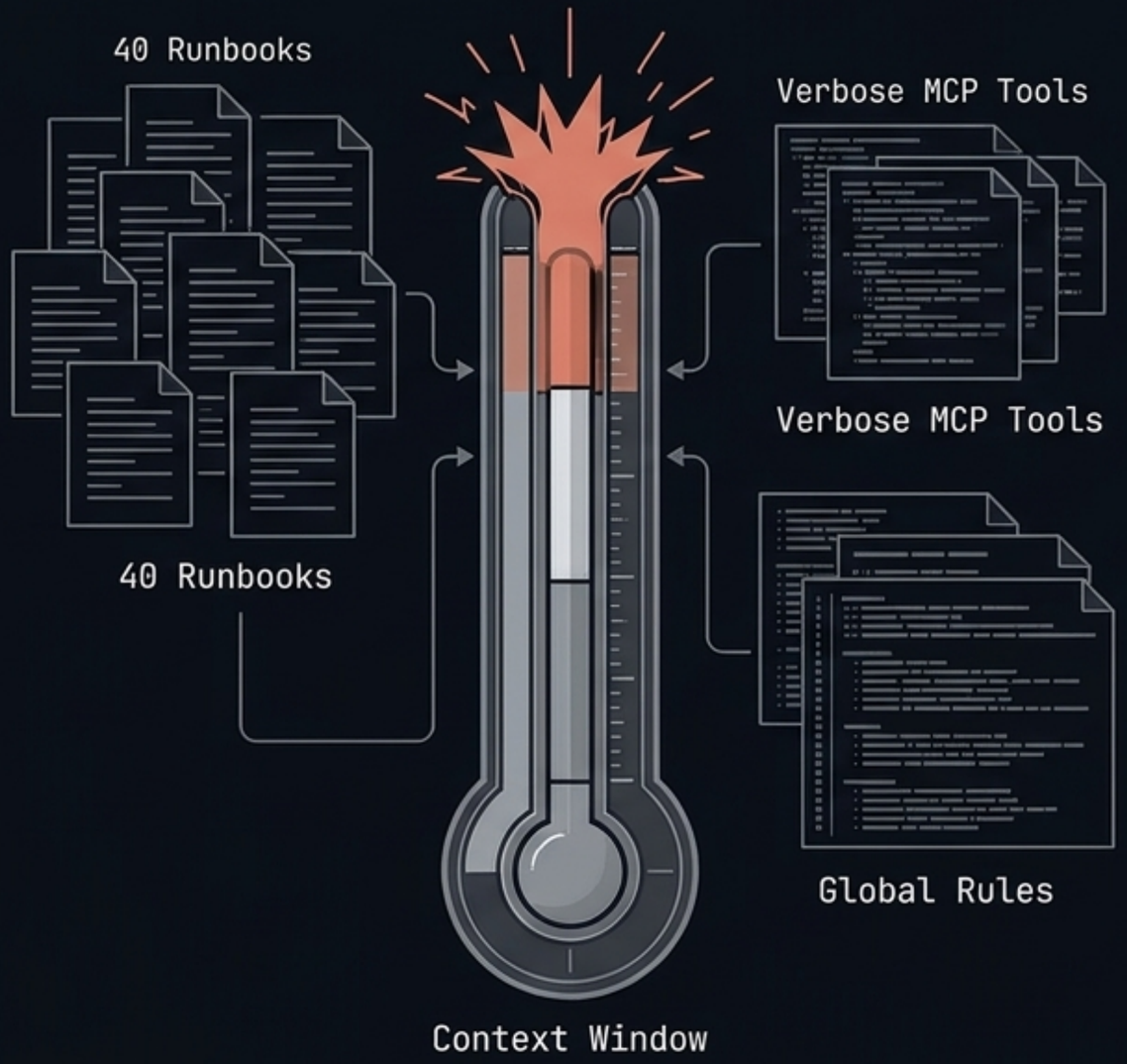




Spec as PR Template: Every pull request links or embeds the IDD spec. Reviewers verify implementation fidelity against the spec, preserving the architectural why in version history.

Curate Instruction Libraries (e.g., auth-api.md, crud-service.md) so new engineers ship compliant services from day one.

The Threat of Context Bloat



The Problem

Dumping procedural runbooks and verbose schemas into AGENTS.md or global context destroys agent performance.

The Symptoms

1

Reasoning Decay:

Accuracy drops as the LLM loses signal in the noise.

2

Latency Spikes:

Time-to-first-token balloons unnecessarily.

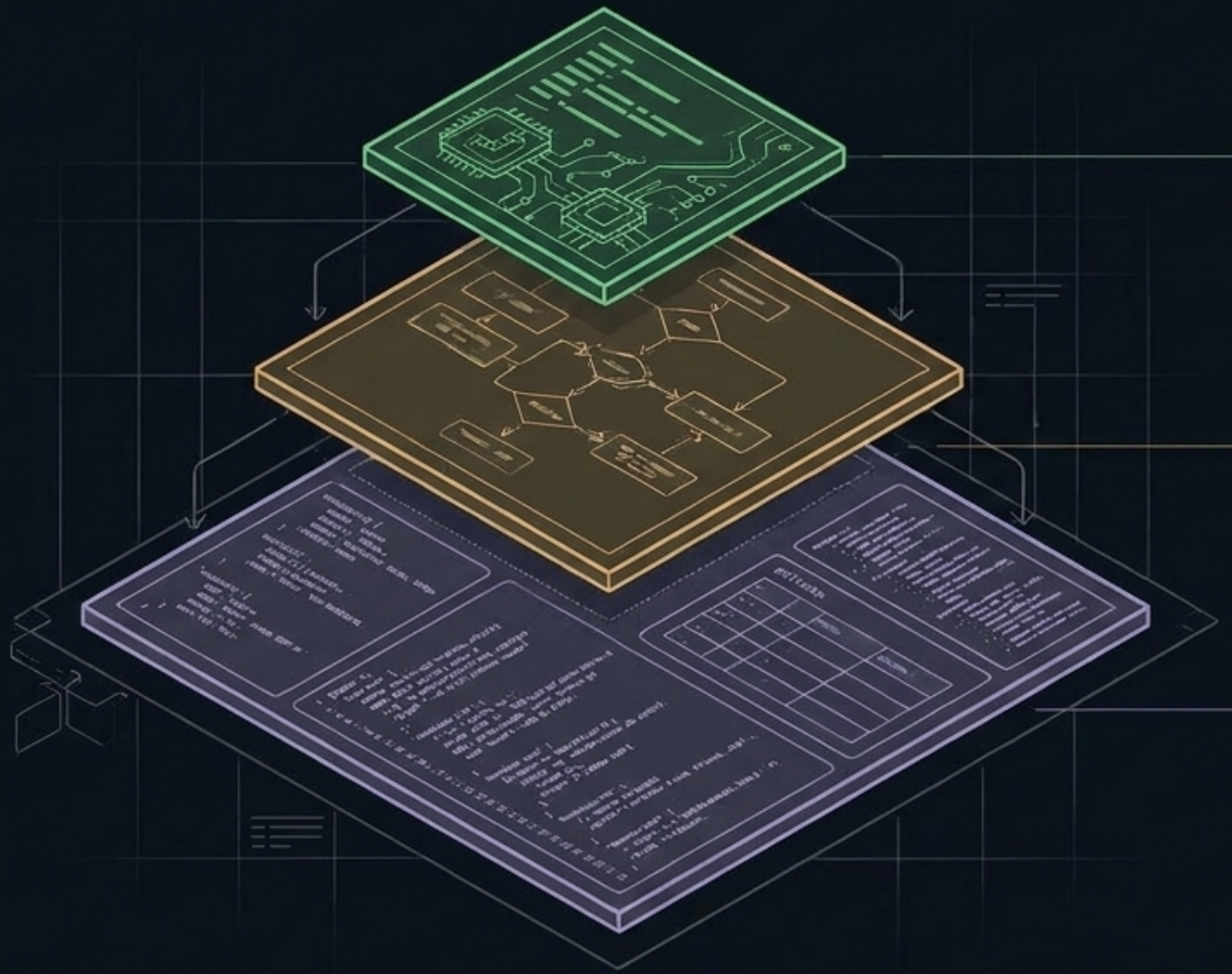
3

Token Waste:

Prompt-cache costs skyrocket on entirely unrelated tasks.

We need a mechanism to give agents infinite capabilities without infinite context...

SKILL.md: 3-Level Progressive Disclosure



Level 1 (Discovery & Catalog):
YAML frontmatter only. Loaded automatically at session startup.

Level 2 (Activation & Instructions):
Full Markdown body. Loaded only when user intent semantically matches.

Level 3 (Execution & Deep Reference):
CLI scripts and API tables. Read or executed strictly on demand.

Standard: Governed by the open Agent Skills standard (agentskills.io).

Skill Activation: Discovery vs. Instructions

Level 1 - Discovery

~100 tokens / skill

Content: YAML frontmatter (name + description).

Critical Authoring Rule: The description is the sole semantic trigger. Vague descriptions fail. Write keyword-rich trigger conditions (e.g., 'trigger when editing service.yaml or debugging 403 ingress').

```yaml
name: debug-ingress
description: "trigger when editing service.yaml
              or or debugging 403 ingress"
tags: [k8s, network, debug]
```

Level 2 - Instructions

< 5,000 tokens recommended

Content: The SKILL.md body.

Function: Gives the agent the step-by-step procedural workflow only after the semantic trigger is hit.

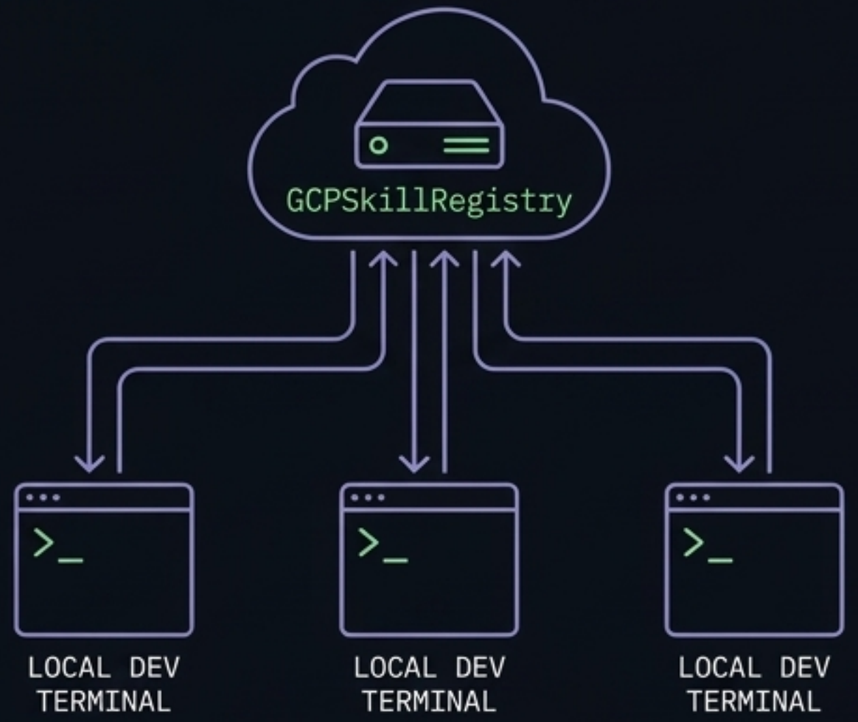
Level 3 Execution & Enterprise Governance

Level 3 Mechanics (Zero Tokens)

Move bulky API tables into `references/` and deterministic validation into `scripts/`. Agents read files or execute CLI tools on demand, costing 0 context window tokens until invoked.

Enterprise Governance

Platform teams publish versioned skills centrally using `GCPSkillRegistry` on Google Cloud, enforcing IAM access, provenance, and pinned rollouts across agents and CLIs.



Critical Rule:

Never use interactive input() or TTY prompts inside skill scripts. Autonomous agents run headlessly and will hang.

The Trifecta of Agent Control



Control is not achieved by writing a louder prompt.
It is achieved by placing the right constraint at the right layer.

Always-On Workspace Rules

Mechanism

Advisory LLM context for repository architecture, naming conventions, and styling.

Antigravity Syntax

`.agents/rules/*.md`. Requires strict `snake_case` triggers (`always_on`, `model_decision`, `glob`, `manual`). Max ~20k-token budget total.

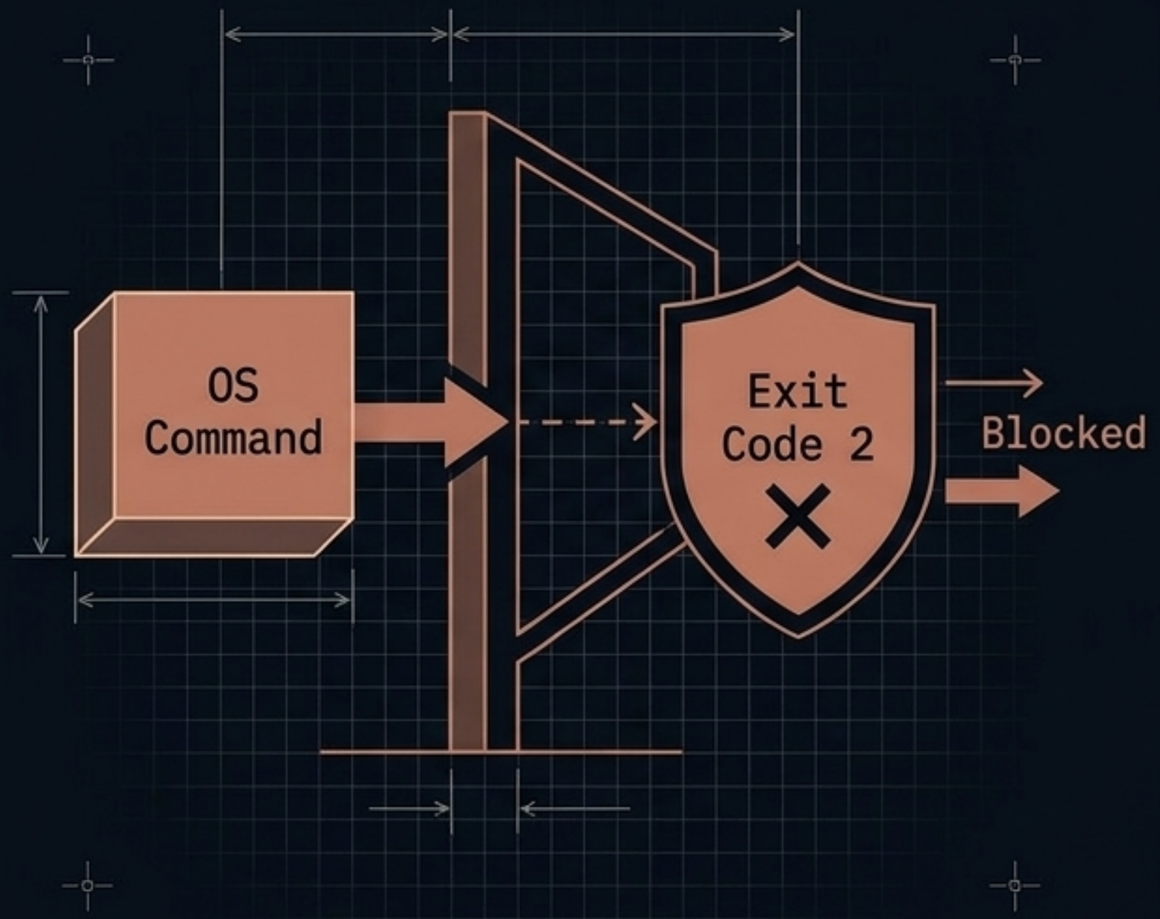
> **Gotcha Warning:** camelCase alwaysOn in Antigravity YAML is silently discarded!

Claude Code Syntax

`.claude/rules/*.md` using YAML globs or `@path` imports in `CLAUDE.md`.

Limitation: Rules are advisory prompt tokens. Under heavy context pressure, an LLM can misinterpret or override them.

Deterministic Hooks: Zero-Exception Control



Mechanism:
100% deterministic OS-level enforcement. Security-critical guardrails must live here, not in prompt rules.

Lifecycle Triggers:
PreToolUse, PostToolUse, Stop, UserPromptSubmit.

The Power of Exit Code 2:
A PreToolUse hook returning exit code 2 deterministically blocks the tool call at the OS/harness layer 100% of the time. (e.g., blocking rm -rf or .env file commits).

Post-Execution:
PostToolUse runs formatters (ruff/tsc); Stop verifies tests pass before yielding control back to the human.

Implementation Matrix: Antigravity vs. Claude Code

Mechanism	Antigravity (Google)	Claude Code
Rules	AGENTS.md, .agents/rules/ with strict snake_case triggers.	CLAUDE.md, .claude/rules/ with YAML globs.
Skills	.agents/skills/ + GCPSkillRegistry	.claude/skills/
Hooks	hooks.json lifecycle scripts	settings.json hooks
Alignment	/plan (executable) & /grill-me	Plan Mode (Shift+Tab)

End-to-End Synthesis: The Spec-to-Eval Automation Pipeline

Intent:
Developer drafts IDD Spec (Phases 1-2).

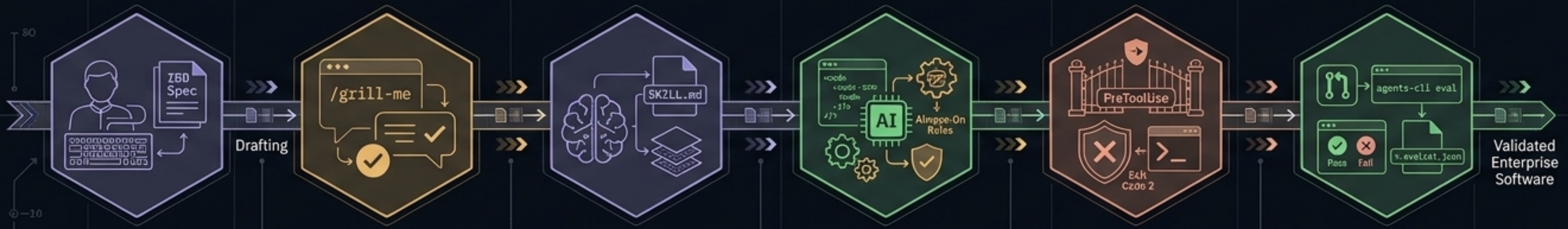
Alignment:
Evaluated via `/grill-me` design interview.

Context:
Intent semantically triggers a Level-2 `SKILL.md` (Progressive Disclosure).

Execution:
AI generates code; guided by Always-On Rules.

Guardrails:
PreToolUse hook validates OS commands.

Evaluation:
Automated PR gating via `agents-cli eval` run against `*.evalset.json`.



Intent:
Developer drafts IDD Spec (Phases 1-2).

Alignment:
Evaluated via `/grill-me` design interview.

Context:
Intent semantically triggers a Level-2 `SKILL.md` (Progressive Disclosure).

Execution:
AI generates code; guided by Always-On Rules.

Guardrails:
PreToolUse hook validates OS commands.

Evaluation:
Automated PR gating via `agents-cli eval` run against `*.evalset.json`.

Synthesis Statement:

From unstructured human thought to mathematically validated enterprise software.

Enterprise Release & Final Takeaways

CI/CD Gotcha

Headless Coding Agents & CI Runners MUST invoke agents-cli with the --yes flag and explicit CLI parameters.

Without this, non-TTY subprocesses will hang indefinitely waiting on stdin prompts.

- [ ] Do you write 6-part specs before generating code?
- [ ] Are your agent skills utilizing 3-Level Progressive Disclosure to prevent bloat?
- [ ] Are your strict security guardrails enforced via PreToolUse Hooks instead of LLM Rules?
- [ ] Are your specs wired into CI via agents-cli eval run?

Stop prompting. Start engineering.