happiest minds
AI FIRST. AGILE ALWAYS.

Rel(AI)Build
ENTERPRISE AI DEVELOPMENT FRAMEWORK

Rel(AI)Build

From Ad-Hoc AI Pilots to a **Governed Engineering Standard**

A Complete Enterprise Framework for
Agentic Software Development

A Complete Enterprise Framework for Agentic Software Development

Orchestration · Governance · Knowledge · Deployment · Observability

7	14	237+	53+	40-60%
IDEs	Stacks	Agents	Team Packs	Token Savings

Table of Content

Executive Summary

LLM coding agents — Cursor, Claude Code, GitHub Copilot, Codex, Windsurf, Kiro — have made autonomous code generation widely available. But they share a fundamental gap: the harness gives the model broad authority through a single configuration file, with no integrity, no process, no portability, and no audit trail. For a single developer exploring, that's fine. For an enterprise with regulatory obligations, multiple teams, and a heterogeneous tool fleet, it's a liability.

Rel(AI)Build is a complete enterprise framework that turns ad-hoc AI pilots into a governed, repeatable engineering standard. It is not another orchestration layer, and it does not replace the harness. Instead, it treats agent definitions as a first-class software supply chain - versioned, content-hashed, tamper-evident, policy-checked — and wraps the entire agentic development lifecycle in deterministic controls that cannot be talked out of by a persuasive prompt.

Author once. Govern centrally. Deploy everywhere. Observe everything."

The framework spans five pillars:

1. **Orchestration:** A hub orchestrator coordinates six specialist agent teams through an 8-phase AI-DLC pipeline, from JIRA story to signed PR.
2. **Governance:** SHA-256 integrity, HMAC lockfiles, five-tier permissions, attack-derived blocklists, and phase-gated workflows with mandatory human-in-the-loop gates.
3. **Knowledge:** 700+ curated skill recipes, 500+ pre-distilled knowledge shards, and stack-specific profiles loaded just-in-time — not bulk-injected.
4. **Deployment:** Define once in canonical YAML+Markdown, auto-compile to seven IDE-native formats — Cursor, Claude Code, VS Code/Copilot, Codex, Windsurf, OpenCode, Kiro.
5. **Observability:** Immutable hash-chained audit log, per-agent token attribution, cycle-time dashboards, and tokenization-drift detection.

Metric	Value
Agent Definitions	237+
Skill Recipes	200+
Knowledge Shards	100+
Pre-Composed Team Packs	53
Technology Stacks	10 (Angular, React, Next.js, Vue, Spring Boot, .NET, Python, PHP, SST,
IDE Targets	7 (Cursor, Claude Code, Copilot, Codex, Windsurf, OpenCode, Kiro)
Token Reduction	Up to 80% vs. naive harness usage

The Challenge

Why AI Harnesses Alone aren't Enough

A modern LLM coding harness is, from a governance standpoint, remarkably simple: the user supplies a natural-language configuration — CLAUDE.md,.cursorrules, AGENTS.md — and the tool injects it into the model's context. The model then plans, edits files, and runs shell commands with broad authority. This minimal interface is what made these tools succeed: there is almost nothing to learn.

The same minimalism is a liability in enterprise settings. Three gaps recur across every harness:

Unmanaged Configuration: A cursorrules file encoding hard-won architectural conventions is copy-pasted between repositories, edited inconsistently, and trusted without provenance. There is no notion of "this is the approved version" or "who changed it and when." Yet this file directly steers an agent with write and execute access.

Unbounded Execution: No enforced requirement → implementation → test trace, no mandatory review checkpoint, and no hard cap on self-correction loops - a known runaway-cost failure mode. NIST AI 600-1 identifies "autonomous action without human oversight" a primary generative AI risk.

No Portability: Each harness has its own configuration dialect, tool-permission syntax, and file layout. An organization using multiple IDEs must re-author and re-maintain the same expertise N times.

Harness	Config File(s)	Tool Permission	Scoping
Cursor	.cursor/rules/*.mdc	implicit	glob / file
Claude Code	CLAUDE.md, .claude/agents/*	comma string	project
Copilot (VS Code)	*.agent.md, instructions	YAML array	file / PR
Codex	root AGENTS.md	-	file
Windsurf	.windsurfrules	-	file
Kiro	steering files, specs	hooks/skills	project

Rel(AI)Build doesn't replace Copilot, Cursor, Claude Code, or Kiro — it makes them enterprise-ready.

Introducing Rel(AI)Build

The Four Motions

Rel(AI)Build operates through four sequential motions, each building on the last:

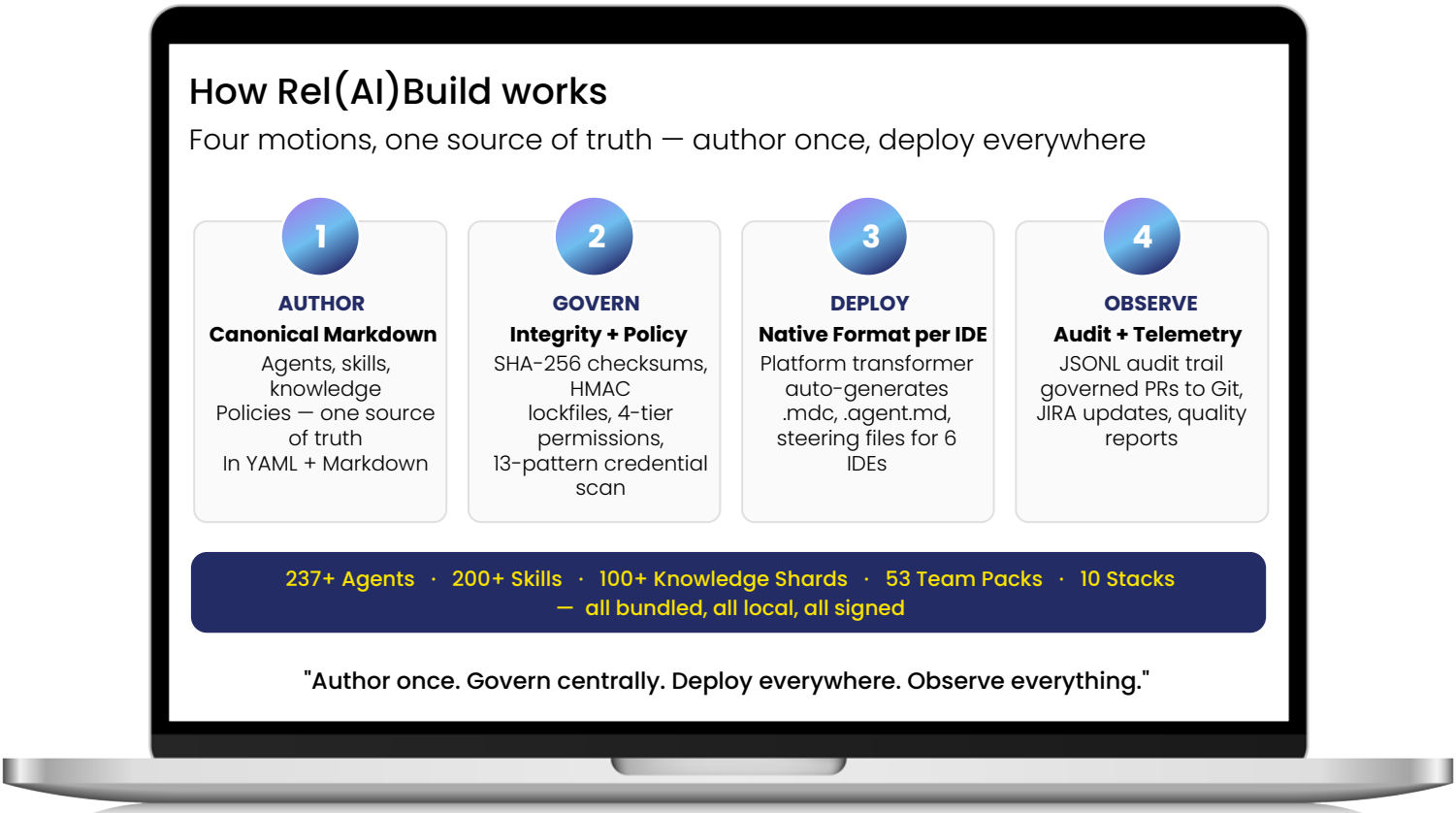


Figure 1: The Four Motions — Author, Govern, Deploy, Observe

1. Author — Canonical Markdown

- Agents, skills, knowledge shards, profiles, and workflows are authored in a single canonical format (YAML + Markdown).
- Five resource types separate concerns that bare configs conflate: Agents (who), Skills (how), Knowledge (what to know), Profiles (standing context), and Workflows (when).

2. Govern — Integrity + Policy

- SHA-256 checksums on every resource. HMAC-stamped lock files for tamper detection.
- Five-tier permission model enforced at install time (fail-closed).
- 13-pattern credential scanner.
- Attack-derived command blocklist. Phase-gated workflows with mandatory HITL gates.
- Tokenization-drift detection with Jaccard similarity.

3. Deploy — Native Format per IDE

- The platform transformer auto-generates. mdc, .md, .agent.md, and other steering files for seven IDEs. Per-platform tool-syntax translation (comma-string vs YAML array vs object).
- Static-content-first ordering for prompt -cache efficiency.
- All governance properties hold uniformly across the fleet.

4. Observe — Audit + Telemetry

- Append-only hash-chained JSONL audit log with per-entry SHA-256.
- Per-agent token attribution and cost tracking. Cycle-time dashboards and throughput analytics.
- Governed PRs to Git with JIRA integration.
- Tokenisation drift alerts when prompts move off their reviewed baseline.

The Agentic Development Lifecycle

Architecture & Orchestration

Rel(AI)Build is structured as a control plane with three layers: an authoring & distribution plane (a canonical registry with content hashes), a compilation plane (a normalizer + cache-friendly ordering + a platform transformer), and a runtime governance plane (deterministic helpers for phase state, delegations, and traceability).

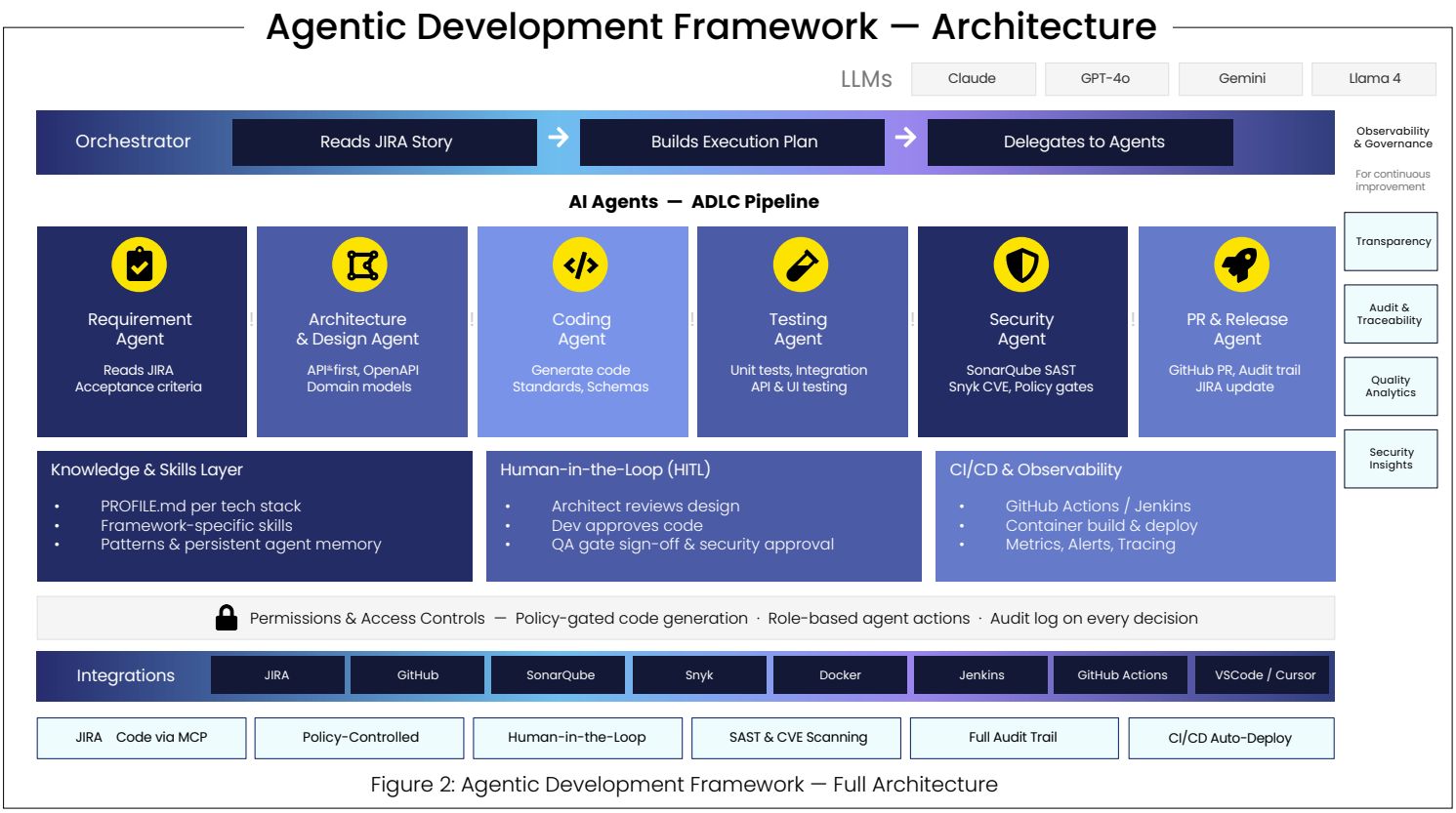


Figure 2: Agentic Development Framework — Full Architecture

The hub orchestrator reads a JIRA story, builds an execution plan, and delegates to six specialist agent teams across a sequential ADLC pipeline: Requirement Agent, Architecture & Design Agent, Coding Agent, Testing Agent, Security Agent, and PR & Release Agent. Each specialist has a declared permission tier, tool allowlist, and glob-scoped context.

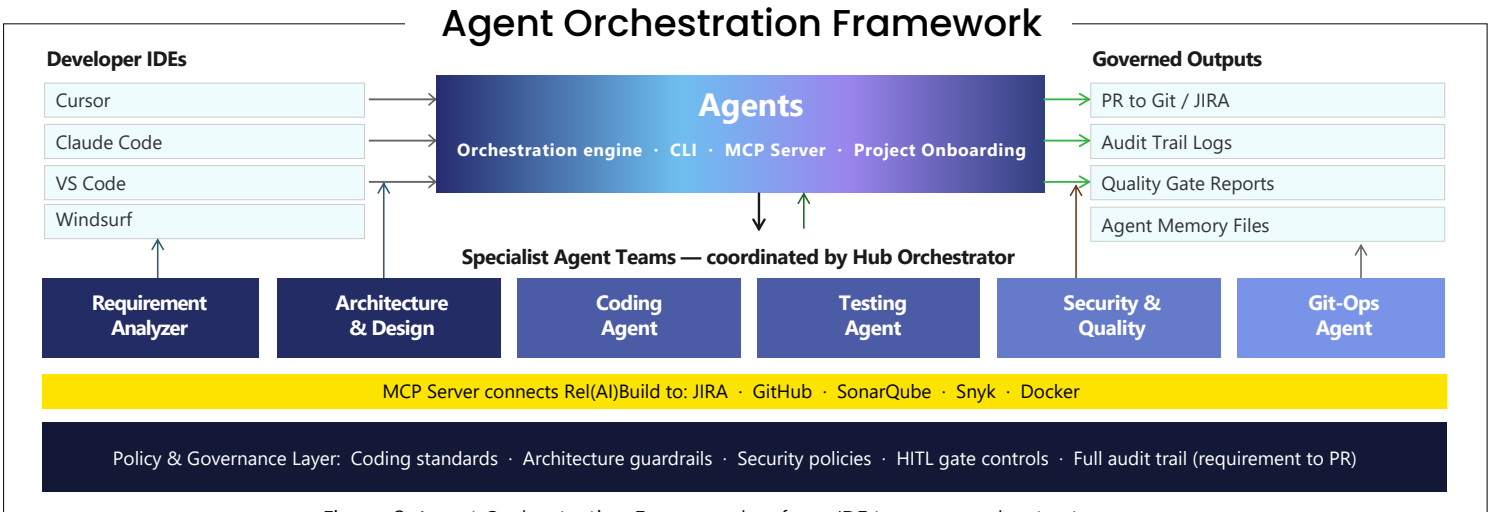


Figure 3: Agent Orchestration Framework — from IDE to governed output

What Gets Installed: A Governed Team

One install command deploys a complete, governed engineering team. 53 pre-composed team packs span 10 technology stacks. Every agent is classified into a permission tier (orchestrator, specialist, operations, or read only), signed with SHA-256, and scoped to specific files via glob patterns. This isn't a prompt library — it's a governed engineering team where every agent is classified, scoped, and signed.

Governance & Guardrails

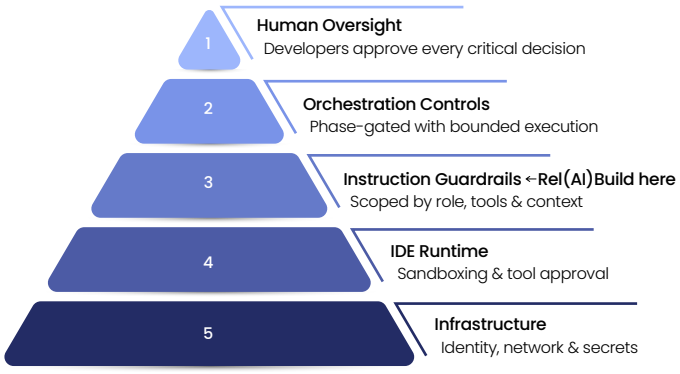
Deterministic Controls That Can't Be Talked Out Of

Every governance mechanism is implemented in ordinary, testable code — hashing, regex matching, a state machine, set arithmetic — not delegated to an LLM. This matters because a non-deterministic component cannot have trustworthy control for a non-deterministic component

Governance & Guardrails

Five layered controls + six governance pillars that make AI-assisted development enterprise-ready.

Five independent layers — Rel(AI)Build owns the layer harnesses leave open



Governance Pillars

1. **Role-Based Delegation** — Scoped permissions per agent role — orchestrators route, specialists execute within boundaries.
2. **Phase-Gated Workflows** — Sequential execution with no-skip rules — agents cannot bypass testing or review phases.
3. **Human-in-the-Loop** — Mandatory approval at critical gates — design review, pre-merge, and fail-safe escalation.
4. **Quality Gates** — Automated code review, testing, and security audit before any code reaches production.
5. **Policy Enforcement** — Bounded retry loops, disciplined context handling, credential scanning, and audit trails.
6. **Review Agents** — Dedicated reviewer and security auditor agents enforce compliance and pattern checks automatically.

Figure 4: Five Independent Governance Layers

Supply-Chain Integrity

Every registry resource has a SHA-256 hash in checksums. json. Install recomputes and aborts on mismatch — blocking context poisoning. The per-project lock file is HMAC-stamped for tamper evidence. An append-only hash-chained audit log provides full provenance: what was installed, by whom, when.

Five-Tier Permission Model

Every agent is assigned exactly one tier. Violations are errors at install/transform time (fail-closed), not runtime warnings. Scribe-tier agents may write only under declared prefixes, with path-traversal defense that normalizes and rejects residual ".." segments.

Tier	Intent	Tools (Summary)	MCP	Shell
readonly	exploration	read / glob / grep / search	read-only	✕
scribe	scoped writes	read + write within declared	✕	✕
operations	git / CI / deploy	read / write / edit / bash / git	declared	✓
specialist	coding / analysis	read / write / edit / bash / glob	declared	✓
orchestrator	top-level	unrestricted	✓	✓

Attack-Derived Blocklist

Regular expressions block shell commands and write targets associated with documented supply-chain attacks — for all tiers, including orchestrators. Each pattern traces to a real incident: npx -y auto-confirm (Nx Console 2026), curl|bash (Codecov 2021), detached-spawn (ua-parser-js 2021), persistence-path writes (xz-utils 2024). On Claude Code, these run as PreToolUse hooks; on other IDEs as post-delegation scans.

Phase-Gated Lifecycle

Feature work runs through 8 sequential phases with four mandatory HITL gates and a 3-iteration cap on auto-fix loops. A deterministic state machine enforces: ordering invariants (starting phase n requires end of n-1), delegation receipts (specialists must be invoked, not bypassed), mandatory review gates, and requirement file test traceability with spec-hash verification.

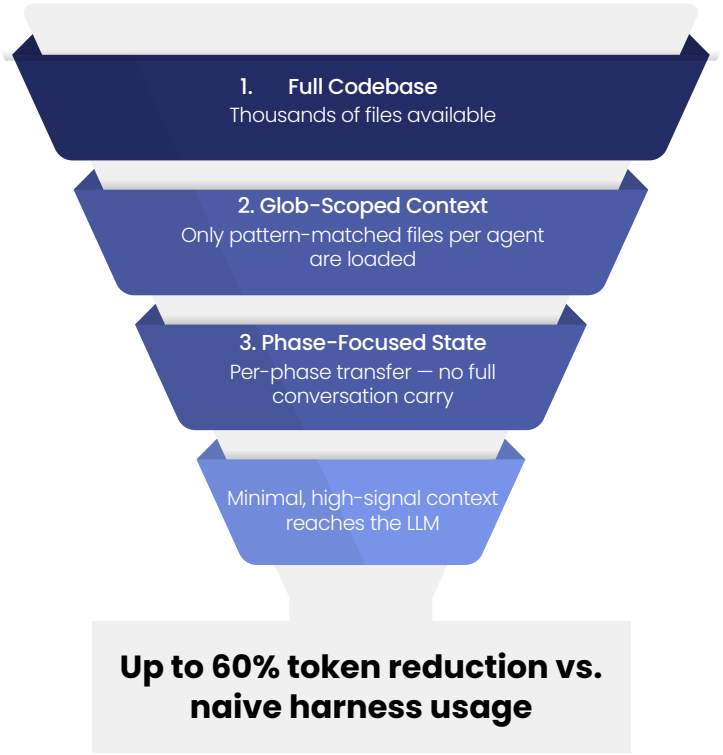
Tokenization Drift Detection

Before compilation, Markdown bodies are canonicalized (CRLF LF, tab spaces, list-marker unification). A Jaccard similarity metric compares against a stored baseline, bucketing risk: LOW (≥0.80), MEDIUM (0.60–0.79), HIGH (<0.60). HIGH drift is a CI-failing signal that a prompt has moved materially off its reviewed baseline.

Token Economics

Engineered for Cost

Token spend is not an afterthought — it's an architectural concern. Rel(AI)Build applies a context funnel that reduces the tokens reaching the LLM by up to 80% compared to naive harness usage:



Up to 60% token reduction vs. naive harness usage

1. **Smart Context Loading**
Glob patterns load only what each agent needs — testing agents see test files, not deployment configs.

2. **Model Tiering**
Right-size the model per task — premium reasoning for architecture, efficient models for formatting & linting.

3. **Bounded Iterations**
3-retry caps on all auto-fix loops prevent runaway token burn; humans take over if unresolved.

4. **Pre-Distilled Knowledge**
100+ curated pattern & troubleshooting shards — agents don't re-analyse the codebase from scratch per interaction.

5. **Context Budget Estimation**
Analyze-tokens CLI estimates installed context size and flags packs exceeding the 2000-token always-on budget.

Figure 5: Token Economics — context funnel and five cost levers

From Requirements to Production

Use Case 1: Greenfield

The 8-phase AI-DLC pipeline turns a CTO's spec into a tested, audit-ready PR — every artifact signed, every gate human-approved. From greenfield requirements (REST API, 1K concurrent users, >80% test coverage), the pipeline produces spec.md, architecture.md with OpenAPI spec, implementation code, unit + functional tests, security audit, and a signed PR with full JSONL audit log.

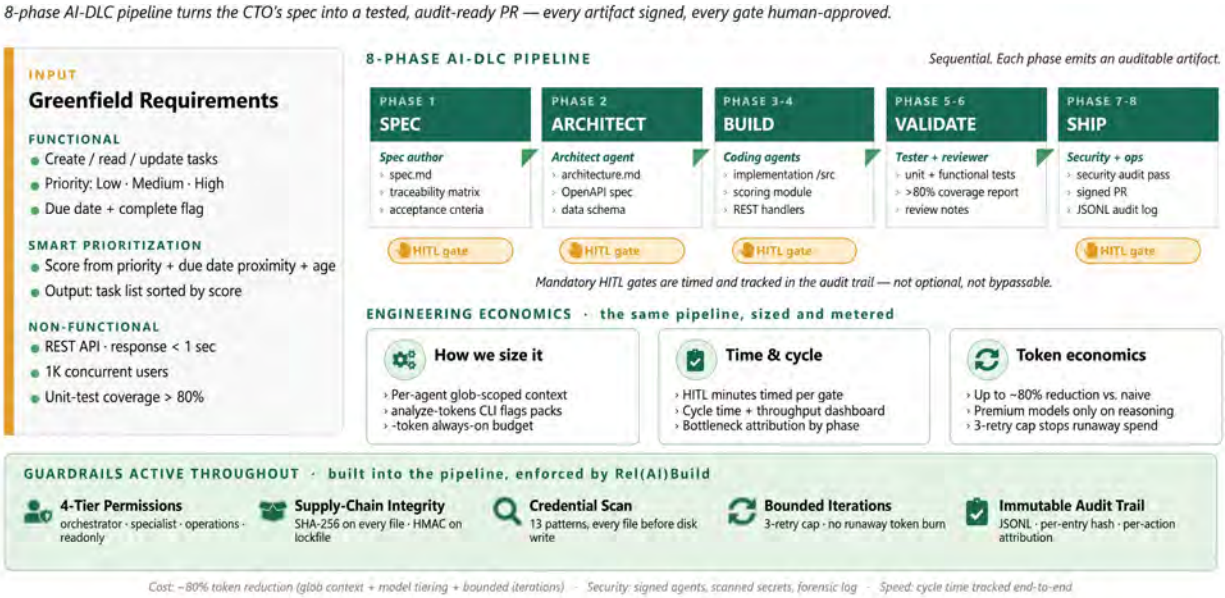


 Figure 6: Greenfield Use Case — 8-phase pipeline from requirements to production-ready API

Use Case 2: Brownfield — Changed Requirements, Zero Drift

The same pipeline handles requirement changes incrementally. Only the changed surface is rebuilt; existing acceptance criteria and tests are preserved. Drift detection verifies at every checkpoint: all new ACs have tasks and file mappings, no file was changed outside acceptance-criteria scope, and 100% AC coverage is verified before HITL approval.

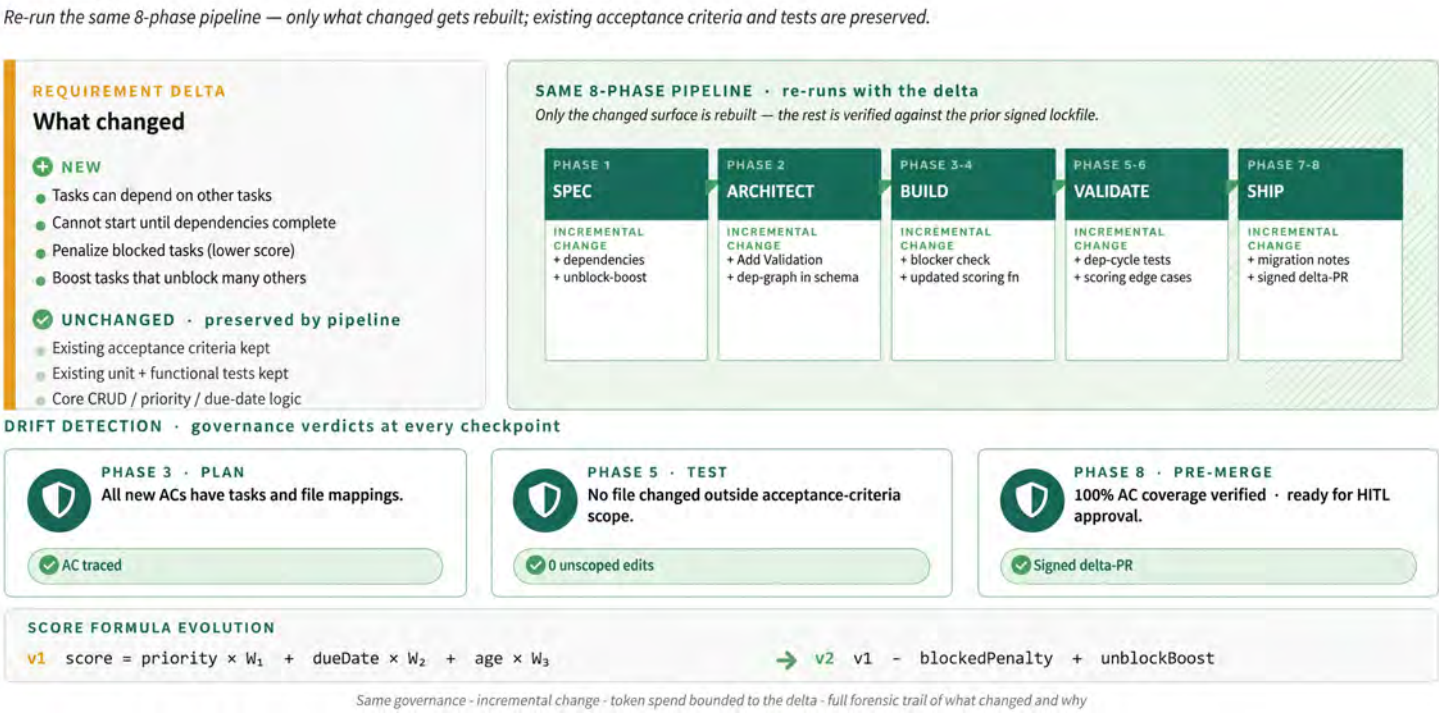


Figure 7: Brownfield Use Case — incremental change with drift detection

Verified Outcomes

Pipeline-produced artifacts demonstrate that every requirement is verifiable: the priority scoring algorithm is implemented with an exact formula; response time <1s is addressed architecturally (async, pre-computed scores, DB indexes); 1000 concurrent users are handled via stateless singletons and scoped services, and unit test coverage is 98% line / 92% branch with 56 tests.

Security & Threat Model

We adopt an attacker who can influence untrusted task inputs, the shared agent-system directory via PRs, and/or dependency suggestions. Mitigations map to OWASP LLM Top 10 and the draft OWASP Agentic-AI Top 10:

Threat	OWASP	Deterministic Mitigation	Residual
Prompt Injection	LLM01	HITL review gates + AC-scope flagging	Medium
Privilege Escalation	LLM06	Fail-closed tier allowlists; traversal defence	Low
Context Poisoning	LLM03	SHA-256 integrity + HMAC lockfile + audit log	Low
Credential Leakage	LLM02/06	13-pattern scanner; redaction; OS-keychain	Low
Uncontrolled Recursion	LLM06	3-iteration cap; bounded delegation depth	Low
Confused Deputy	Agentic SC	Attack-derived blocklist; pkg-manager wrappers	Low-Med

Prompt injection remains medium residual: we do not claim to solve injection. We claim to bound its blast radius with deterministic scope, permission, and integrity controls. The human gate is the backstop.

Credential Handling

A 13-pattern scanner covers GitHub PAT (ghp_), OpenAI (sk-), AWS (AKIA), Stripe, Slack, Snyk, Figma, npm, plus a broad *_TOKEN/SECRET/PASSWORD heuristic. Runtime secrets are fetched from the OS keychain via a shell-free execFile call, not from disk. Pattern-based detection is inherently incomplete; novel formats slip through; that is termed residual risk.

Conclusion

The agent-configuration-and-process layer — the prompts, permissions, and process that turn a general LLM into a coding agent — is currently trusted implicitly and managed informally, even as it directs tools with write and execute authority.

Rel(AI)Build demonstrates that this layer can be governed with the same rigour we apply to code dependencies, and that the governance can be made deterministic (so it is trustworthy) and tool-agnostic (so it scales across a heterogeneous fleet). Content-hashed, tamper-evident, policy-checked agent definitions; fail-closed permissions and attack-derived blocklists; tokenization-drift detection; a phase/traceability state machine that refuses to advance on violation; and token economics engineered from the ground up — all compiled once and deployed to seven harnesses.

The architecture should follow the problem, not the trend. Governance of the configuration and process layer is a solvable, deterministic problem — and solving it is a prerequisite for trusting AI coding agents at enterprise scale.

Author

Padmaraj Madatha

Chief Architect — Office of the CTO, Happiest Minds Technologies

Is a seasoned technology leader who is currently leading the AI Productivity CoE Innovations. With over 26+ years, Padmaraj brings a strong technical skill set coupled with a genuine love for solving real business problems. He has applied a variety of technology and SaaS platforms throughout his career, creating secure, scalable, and sustainable systems.

His experiences have spanned across multiple sectors--healthcare, banking, insurance, edtech and automotive, which indicates his ability to understand the nuances of each domain and understand their compliance needs and provide solutions appropriately.

He has vast amount of experience across technologies which include cloud computing, IoT, machine learning, and more recently quantum computing. It is his ability to simplify very complex technology into simple, usable, and valuable solutions for the customer that makes him unique.

About Happiest Minds Technologies

Happiest Minds Technologies Limited (BSE, NSE: HAPPSTMNDS) is an AI First, customer-centric digital engineering company committed to delivering 'Happiest People . Happiest Customers'. With an integrated approach that spans from chip to cloud, Happiest Minds delivers secure and scalable solutions across product engineering, cybersecurity, analytics , and automation platforms. Happiest Minds brings purpose and precision to every engagement, helping enterprises solve complex business challenges and fast-track their digital evolution across industry sectors such as Banking, Financial Services & Insurance (BFSI), EdTech, Healthcare & Life Sciences, Hi-Tech and Media & Entertainment, Industrial, Manufacturing, Energy & Utilities, and Retail, CPG & Logistics. Happiest Minds has been honored by both the Golden Peacock Awards and the Institute of Company Secretaries of India (ICSI) for its exemplary Corporate Governance practices. Guided by its mission of 'Happiest People . Happiest Customers' and consistently recognized as a great place to work, Happiest Minds is headquartered in Bengaluru, India, with a global presence across the Americas, UK, Europe, Australia, the Middle East, Africa, and Asia.

happiest minds
AI FIRST. AGILE ALWAYS.

www.happiestminds.com

For more information, please write to us at business@happiestminds.com