AI-Native Git Layer — Complete Implementation Report

Author -Rahel Samson

Executive Summary

The **Intent-Driven Governance Layer** is a middleware system built into Roo-Code that provides formal, deterministic control over AI agent actions. It enforces explicit intent declaration, scope boundaries, human authorization, concurrency safety, and immutable mutation traceability.

It directly addresses three systemic problems in AI-assisted development:

Problem	Definition	How This System Addresses It
Cognitive Debt	Silent divergence between what a developer intends and what the AI agent actually executes	The Intent Handshake forces the agent to explicitly declare its objective; constraints and scope are injected into every destructive tool call
Trust Debt	Accumulated opacity where developers can no longer verify what the agent changed or why	An append-only trace ledger records every mutation with canonical file paths, disk-based content hashes, classification, and Git revision identifiers
Context Rot	Model context diverges from actual project state due to stale reads, concurrent edits, or environmental drift	Optimistic locking via read-hash tracking detects and blocks stale writes until the file is re-read

The system was implemented across **4 phases**, producing:

- **20 source files**
- **3 dedicated test suites**
- **49 passing tests**
- A deterministic trace ledger
- Optimistic concurrency control
- Scope-based access enforcement
- Human-in-the-loop authorization
- Failure learning via lesson recording

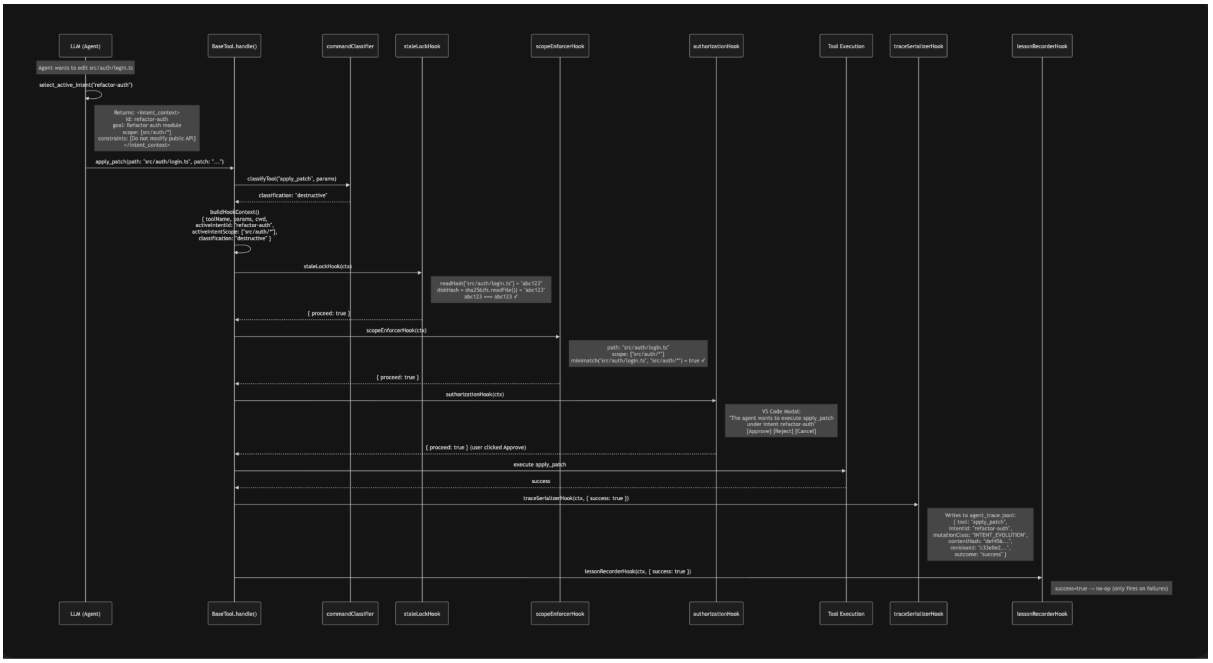
The architecture is deterministic, restart-consistent, and auditable.

---

1. Architecture Overview

The governance layer wraps Roo-Code's `BaseTool.handle()` method with a pre-hook and post-hook enforcement pipeline. Every destructive tool invocation passes through deterministic validation layers before execution, and audit serialization layers after execution.

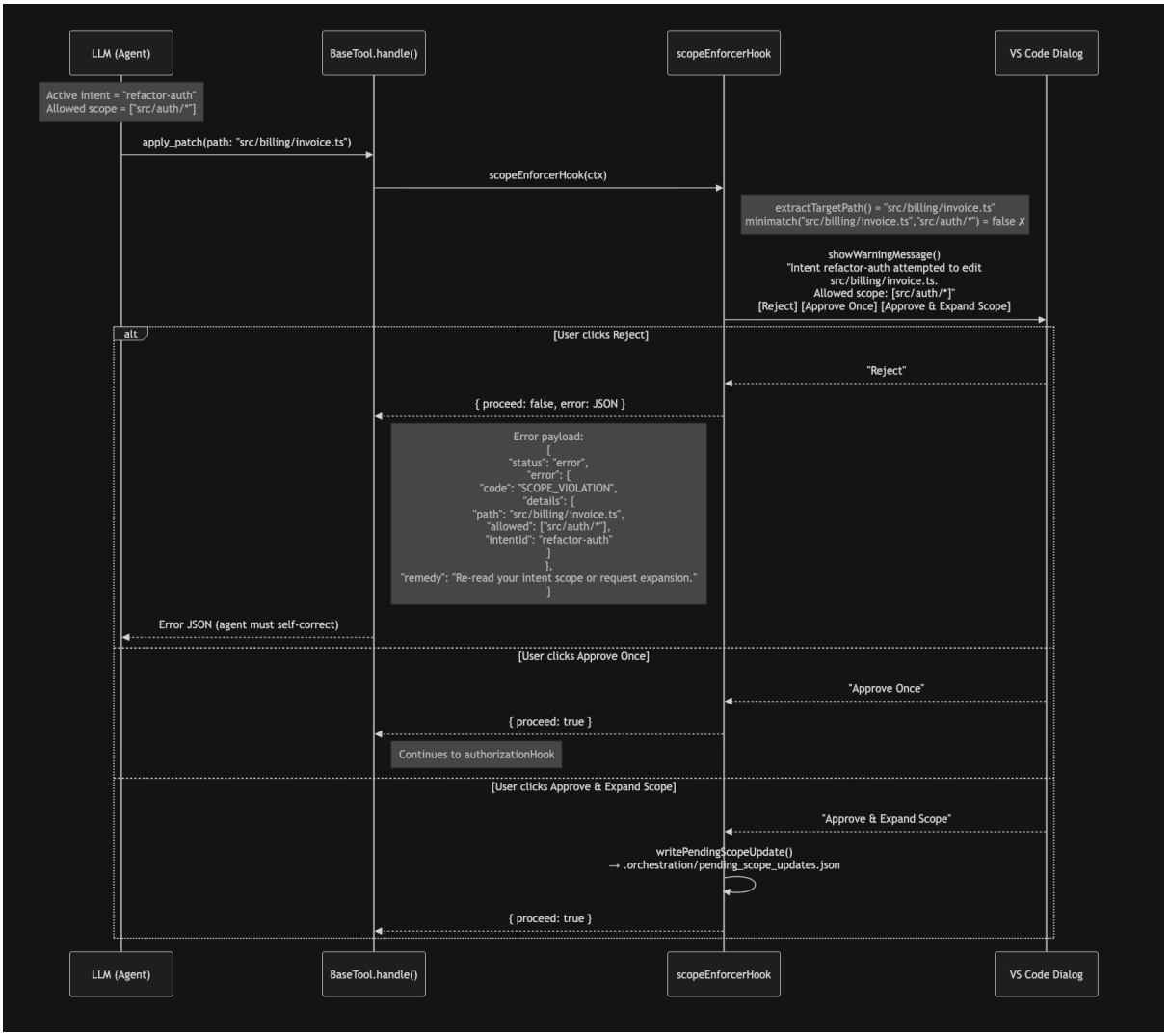
1.1 System Flow — Happy Path



This happy-path flow demonstrates:

- Explicit intent selection before mutation
- Deterministic tool classification
- Optimistic concurrency validation
- Scope enforcement
- Human authorization
- Post-execution ledger serialization
- Conditional lesson recording

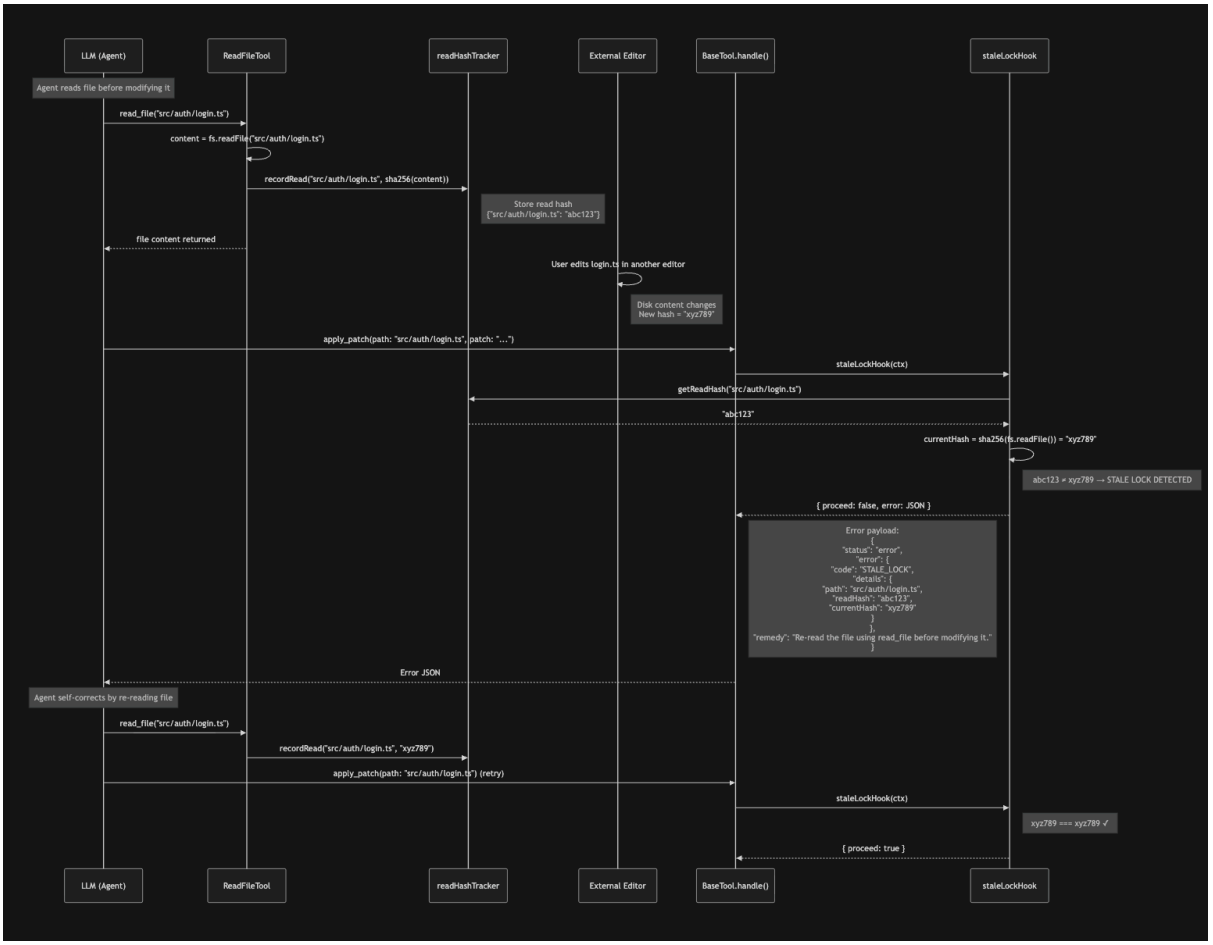
1.2 Failure Path — Scope Violation



This failure path shows:

- Deterministic path extraction
- Minimatch-based scope validation
- Human override options
- Structured error payload returned to the LLM
- Optional scope expansion persistence

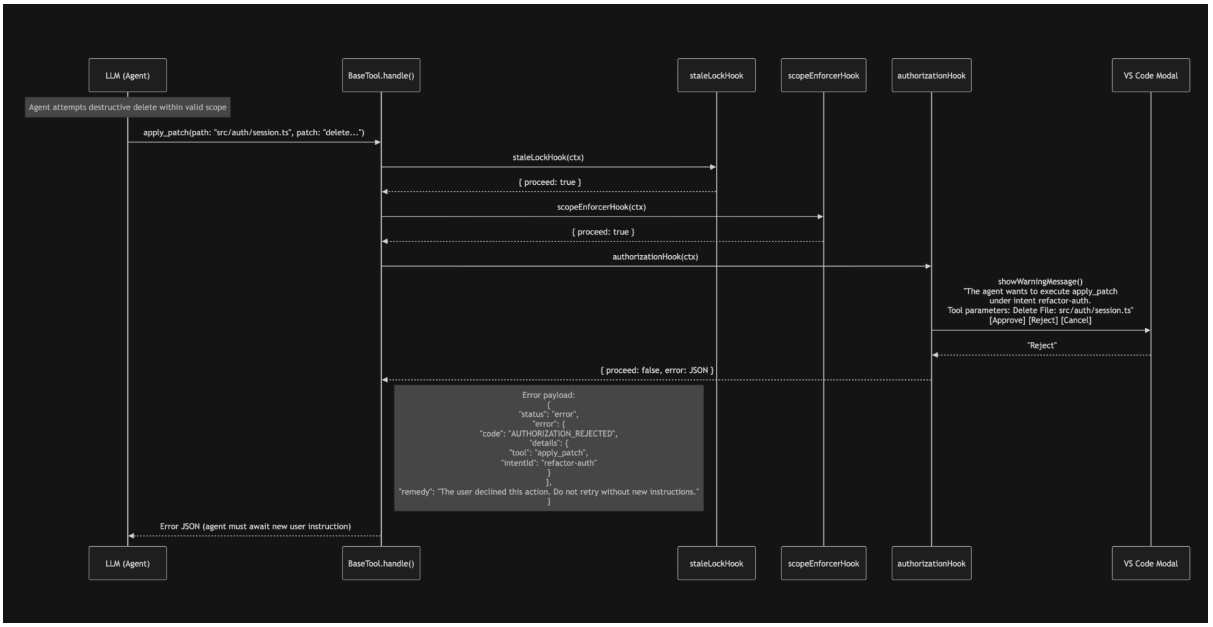
1.3 Failure Path — Stale Lock Detection



This flow demonstrates:

- Raw Buffer hashing (not string hashing)
- Optimistic concurrency validation
- Deterministic stale detection
- Forced re-read before retry
- Safe retry path after state convergence

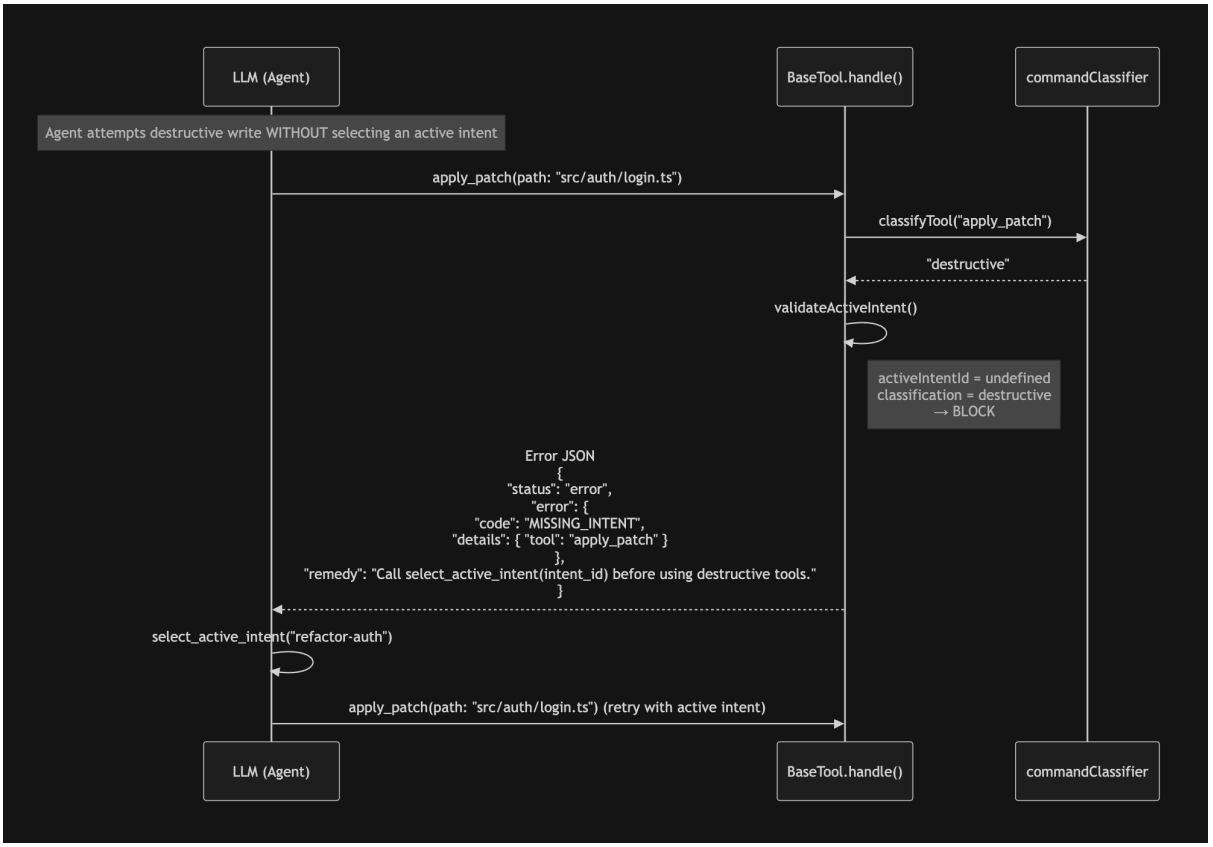
1.4 Failure Path — Authorization Rejection



This flow demonstrates that:

- Even when scope and concurrency validations pass, human authorization remains mandatory for destructive tools.
- The rejection is structured, machine-readable, and actionable.
- The agent must not retry without new user instruction.

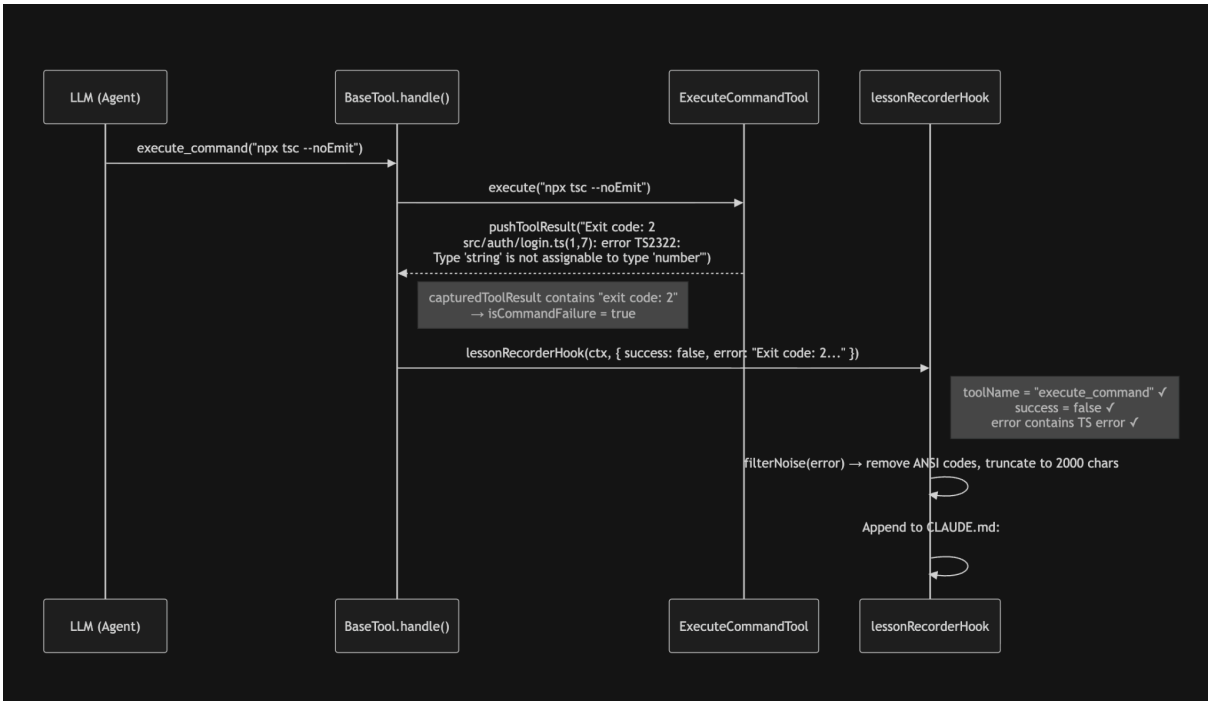
1.5 Failure Path — Missing Intent



This ensures:

- Destructive tools cannot execute without an explicitly selected intent.
- Intent selection is a hard prerequisite.
- The system enforces explicit developer alignment before mutation.

1.6 Post-Hook — Lesson Recording (Command Failure)



This mechanism ensures:

- Command verification failures become persistent project knowledge.
- Errors are deduplicated within a session.
- Noise (ANSI codes, excessive length) is filtered.
- The system compounds failure insight into institutional memory.

---

2. Detailed Schemas

2.1 Intent Configuration — `active_intents.yaml`

Location: `.orchestration/active_intents.yaml` (workspace root)
Owned by: Human developer (manually created or edited)
Read by: `activeIntents.ts` during intent selection; `scopeEnforcer.ts` during scope validation
Update trigger: Manual edit by developer or orchestrator agent

```
intents:
 - id: "refactor-auth"                # string — Unique intent identifier
   goal: "Refactor authentication..."   # string — Injected into agent system prompt
   status: IN_PROGRESS                  # enum: IN_PROGRESS | PAUSED | DONE
```

```
  constraints:                  # string[] — Behavioral constraints
    - "Do not modify public API"
    - "Preserve existing function signatures"
  scope:                        # string[] — Glob patterns defining writable scope
    - "src/auth/*"
```

Field Semantics

Field	Type	Purpose	Consumed By
`id`	`string`	Unique key referenced in trace entries and hook context	`select_active_intent`, `traceSerializerHook`, `scopeEnforcerHook`
`goal`	`string`	Human-readable objective injected into system prompt	`select_active_intent`
`status`	`"IN_PROGRESS" | "PAUSED" | "DONE"`	Only `IN_PROGRESS` intents may be selected	`select_active_intent`
`constraints`	`string[]`	Injected into `<intent_context>` XML block	`select_active_intent`
`scope`	`string[]`	Glob patterns validated via `minimatch`	`scopeEnforcerHook`

The `intentId` selected here propagates deterministically into:

- HookContext
- Trace ledger entries
- Mutation classification
- Failure messages

2.2 Agent Trace Entry — `agent_trace.jsonl`

Location: `.orchestration/agent_trace.jsonl` (workspace root)
Owned by: `traceSerializerHook` (append-only, post-hook)
Read by: `mutationClassifier.ts` on first classification call
Update trigger: Every successful file mutation

```
interface AgentTraceEntry {
  id: string            // UUIDv4 — unique ledger entry
```

```
  timestamp: string       // ISO-8601 — mutation time
  tool: string            // Tool name performing mutation
  intentId: string        // Active intent at mutation time
  mutationClass:
     "INTENT_EVOLUTION"
    | "AST_REFACTOR"
  mutationType:
     "WRITE"
    | "DELETE"
  filePath: string        // Canonical POSIX-relative path
  contentHash: string     // SHA-256 hash of disk content AFTER write
  outcome:
     "success"
    | "error"
  error?: string
  revisionId?: string     // git rev-parse HEAD
  fileSizeBytes?: number
  toolArgsSnapshot?: {
     path: string        // NEVER includes file content
  }
}
```

Example Entries (Real Output)

{"id":"00a0acfb-2f51-4838-8075-c49c1fe0f32f","timestamp":"2026-02-21T12:30:34.890Z","tool":"apply_patch","intentId":"refactor-auth","mutationClass":"INTENT_EVOLUTION","mutationType":"WRITE","filePath":"src/auth/login.ts","contentHash":"226042533640940557 2643dad174f8c0ac164a8a1ec431c9a4a7caf20856d406","outcome":"success","revisionId":"c33e8e26f6e893ef77d0ef729620b4dc29835c67","fileSizeBytes":2847,"toolArgsSnapshot":{"path":"src/auth/login.ts"}}
{"id":"7e60c1d5-b0b6-4131-9ab0-c715bcd33621","timestamp":"2026-02-21T15:06:21.426Z","tool":"apply_patch","intentId":"refactor-auth","mutationClass":"AST_REFACTOR","mutationType":"WRITE","filePath":"src/auth/login.ts","contentHash":"ffc660359e1470a0dcb7d99c80dfd931ff553298911a62bb740b1b5871a56909","outcome":"success","revisionId":"c33e8e26f6e893ef77d0ef729620b4dc29835c67","fileSizeBytes":2843,"toolArgsSnapshot":{"path":"src/auth/login.ts"}}

Note:

- `intentId` directly references `active_intents.yaml`
- `mutationClass` transitions from `INTENT_EVOLUTION` to `AST_REFACTOR`
- `contentHash` is computed from disk, not from tool payload
- Ledger is append-only and immutable

2.3 Hook Context — `HookContext`

Owned by: `HookEngine.buildHookContext()`
Consumed by: All pre- and post-hooks

```
interface HookContext {
  toolName: string
  params: Record<string, unknown>
  cwd: string
  activeIntentId?: string
  activeIntentScope?: string[]
  classification?: ToolClassification
}
```

The HookContext is the single deterministic state object passed through the hook chain.

2.4 Hook Result — `HookResult`

```
interface HookResult {
  proceed: boolean
  error?: string
  reason?: string
}
```

Standard Error Payload Structure

```
{
 "status": "error",
 "message": "Human-readable description",
 "error": {
   "code": "SCOPE_VIOLATION | AUTHORIZATION_REJECTED | STALE_LOCK |
PATH_TRAVERSAL",
   "details": {}
 },
 "remedy": "Actionable instruction for the LLM"
}
```

2.5 Tool Classification

```
type ToolClassification = "safe" | "sensitive" | "destructive"
```

Tool classification is deterministic and occurs before any hook execution. It defines the enforcement pathway for a tool invocation.

Classification	Tools	Hook Behavior

`safe`	`read_file`, `list_files`, `search_files`, `list_code_definition_names`	**Bypasses all hooks** — no enforcement, no authorization dialog
`sensitive`	Reads of `.env`, `.pem`, `*.key`, `id_rsa`, `*secret*`, `*credential*`, `.orchestration/*`	Authorization dialog required, **no scope enforcement**
`destructive`	`write_to_file`, `apply_patch`, `apply_diff`, `execute_command`, `delete_file`	**Full enforcement chain**: stale lock → scope enforcement → authorization → trace serialization

Classification is performed by `commandClassifier.ts` and is purely rule-based. It does not rely on model judgment.

2.6 Mutation Classification

Mutation classification determines whether a file write represents:

- A first mutation under a given intent
- Or a subsequent refinement under the same intent

Class	Cache Lookup	Meaning
`INTENT_EVOLUTION`	Key `"intentId::filePath"` NOT in `priorWrites`	First successful write by this intent to this file
`AST_REFACTOR`	Key `"intentId::filePath"` EXISTS in `priorWrites`	Subsequent refinement of already-mutated file

Deterministic Key Format

${intentId}::${canonicalFilePath}

Cache Lifecycle

1. On first call to `classifyMutation()`:
    - `agent_trace.jsonl` is read

    - All successful `WRITE` mutations are loaded into `Set<string>`
2. After each successful trace write:
    - `recordWrite(intentId, filePath)` updates the in-memory set
3. On extension restart:
    - Cache resets to `null`
    - Rehydrated from ledger on next classification call

This ensures:

- Restart consistency
- Ledger-derived classification
- No in-memory-only source of truth

---

3. Hook-by-Hook Specification

Each hook below includes:

- Trigger conditions
- Inputs
- Outputs
- State dependencies
- Deterministic guarantees

3.1 Pre-Hook 1: `staleLockHook` — Optimistic Concurrency Control

File: `src/hooks/staleLockHook.ts`
Addresses: Context Rot

Purpose

Prevents overwriting a file if it has changed since the agent last read it.

Trigger Conditions

- Tool classification === `destructive`
- Tool targets a file path
- A prior `read_file` call recorded a hash for that path

Inputs

- `HookContext`
- `params.path`
- `readHashTracker`

Outputs

- `{ proceed: true }`
- OR `{ proceed: false, error: STALE_LOCK JSON }`

Failure Condition

sha256(currentDiskBuffer) !== storedReadHash

State Dependency

`readHashTracker` — module-level `Map<string, string>`

State Management

- `recordRead(path, hash)` → called after `read_file`
- `getReadHash(path)` → returns stored hash

- `clearRead(path)` → called after successful write
- `clearAllReads()` → resets tracking on restart

Deterministic Guarantee

- Hash computed from raw `Buffer`
- Never hashes UTF-8 strings
- Always hashes disk content at time of validation

3.2 Pre-Hook 2: `scopeEnforcerHook` — Path-Based Access Control

File: `src/hooks/scopeEnforcer.ts`
Addresses: Cognitive Debt

Purpose

Ensures destructive mutations remain within declared intent boundaries.

Trigger Conditions

- Tool classification === `destructive`
- Path can be extracted from parameters

Inputs

- `HookContext`
- `activeIntentScope`
- `params`
- `.intentignore`

Outputs

- `{ proceed: true }`
- OR `SCOPE_VIOLATION`
- OR `PATH_TRAVERSAL`

Writes

- `.orchestration/pending_scope_updates.json` (only on user-approved expansion)

Path Validation Pipeline

1. `extractTargetPath(params)`
2. `normaliseAndValidatePath(path, cwd)`
3. Reject if path escapes workspace root
4. Apply `.intentignore`
5. `minimatch` against each scope glob

Deterministic Guarantee

- All paths canonicalized to POSIX
- Scope evaluation independent of OS path separators
- Traversal prevention via `path.resolve()`

3.3 Pre-Hook 3: `authorizationHook` — Human-in-the-Loop Gate

File: `src/hooks/authorizationHook.ts`
Addresses: Trust Debt

Purpose

Ensures destructive and sensitive actions require explicit human approval.

Trigger Conditions

- classification === `destructive`
- OR classification === `sensitive`

Inputs

- `HookContext`

Outputs

- `{ proceed: true }`
- OR `AUTHORIZATION_REJECTED`

Dialog Content

[Intent Governance] The agent wants to execute "{toolName}"
under intent "{activeIntentId}".

Tool parameters: {formatted params}

Do you approve this action?
[Approve] [Reject] [Cancel]

Deterministic Guarantee

- Always shown for destructive tools
- No state caching
- No auto-approval

3.4 Post-Hook 1: `traceSerializerHook` — Immutable Audit Ledger

File: `src/hooks/traceSerializerHook.ts`
Addresses: Trust Debt

Purpose

Serializes every successful file mutation into an append-only ledger.

Trigger Conditions

- Tool in `WRITE_TOOLS` OR `DELETE_TOOLS`
- `outcome.success === true`

Inputs

- `HookContext`
- Tool outcome
- Disk file content
- Git revision

Writes

- `.orchestration/agent_trace.jsonl`

Processing Chain

1. Extract canonical path
2. Read file from disk
3. Compute SHA-256 hash
4. Classify mutation
5. Get Git revision
6. Append JSON line
7. Update mutation cache

8. Clear stale lock read entry

Deterministic Guarantee

- Hash from disk content only
- Append-only ledger
- No in-place edits
- Classification derived from ledger state

3.5 Post-Hook 2: `lessonRecorderHook` — Failure Learning

File: `src/hooks/lessonRecorderHook.ts`
Addresses: Cognitive Debt

Purpose

Persists verification failures into project memory.

Trigger Conditions

- `toolName === "execute_command"`
- `success === false`
- Non-zero exit code detected

Inputs

- `HookContext`
- `outcome.error`

Writes

- `CLAUDE.md` (workspace root)

Deduplication

- `Set<string>` keyed by `sha256(errorMessage)`

Processing Pipeline

1. Validate command failure
2. Dedup check
3. Remove ANSI codes
4. Truncate to 2000 chars
5. Extract command
6. Append formatted lesson block

Deterministic Guarantee

- No duplicate lessons per session
- Append-only file writes
- No mutation of previous lessons

4. File Inventory with Ownership

This section enumerates all governance-layer files, their phase of introduction, and explicit read/write responsibilities.

Core Hook System (`src/hooks/`)

File	Phase	Writes To	Read By	Purpose
`types.ts`	2	—	All hooks	Type definitions: `HookContext`, `HookResult`,

				`PreHookFn`, `PostHookFn`
`HookEngine.ts`	2	—	`BaseTool.ts`	Orchestrates pre/post hook chains, builds context deterministically
`commandClassifier.ts`	2	—	`HookEngine.ts`	Classifies tools as `safe`, `sensitive`, or `destructive`
`scopeEnforcer.ts`	2	`pending_scope_updates.json`	`HookEngine.ts`	Glob-based scope enforcement and path traversal prevention
`authorizationHook.ts`	2	—	`HookEngine.ts`	VS Code modal requiring explicit user consent
`toolError.ts`	2	—	All blocking hooks	Structured JSON error builder with `errorCode` and `remedy`
`intentIgnore.ts`	2	—	`scopeEnforcer.ts`	`.intentignore` parsing (similar to `.gitignore`)
`pendingScopeUpdates.ts`	2	`pending_scope_updates.json`	Orchestrator agent	Persists scope expansion proposals
`pathNormalize.ts`	3	—	`scopeEnforcer.ts`, `traceSerializerHook.ts`	Canonical POSIX path normalization
`contentHash.ts`	3	—	`traceSerializerHook.ts`, `readHashTracker.ts`, `staleLockHook.ts`	Raw-buffer `sha256(content)` utility

traceSchema.ts	3	—	traceSerializerHook.ts, mutationClassifier.ts	AgentTraceEntry interface + WRITE_TOOLS/DELETE_TOOLS constants
mutationClassifier.ts	3	— (in-memory cache)	traceSerializerHook.ts	Deterministic INTENT_EVOLUTION vs AST_REFACTOR classification
traceSerializerHook.ts	3	agent_trace.jsonl	HookEngine.ts	Append-only audit ledger writer
readHashTracker.ts	4	— (in-memory Map)	staleLockHook.ts, ReadFileTool.ts	Tracks SHA-256 of last-read file content
staleLockHook.ts	4	—	HookEngine.ts	Blocks stale writes via optimistic concurrency
lessonRecorderHook.ts	4	CLAUDE.md	HookEngine.ts	Records command verification failures
index.ts	—	—	External consumers	Barrel export for governance modules

---

Modified Core Files

File	Change	Rationale
src/core/tools/BaseTool.ts	handle() wraps pushToolResult, orchestrates pre/post hooks	Required to capture command failures that do not throw exceptions
src/core/tools/ReadFileTool.ts	Calls recordRead() after reading file content	Enables optimistic concurrency tracking

`src/core/context/activeIntents.ts`	Loads YAML, manages active intent state	Implements Phase 1 intent selection and context injection

---

Test Suites

File	# Tests	Coverage
`src/hooks/__tests__/HookEngine.spec.ts`	Phase 2	Hook chain execution, fail-fast behavior, context building, classification
`src/hooks/__tests__/phase3.spec.ts`	Phase 3	Trace serialization, mutation classification, path normalization, hashing
`src/hooks/__tests__/phase4.spec.ts`	Phase 4	Read hash tracking, stale lock validation, lesson recording, deduplication

---

5. Design Decisions with Justification

Decision	Rationale	Problem Domain Reference
YAML-based intents over SQLite	Human-editable, version-controllable (`git diff` friendly), no binary dependency	Reduces **Cognitive Debt** by keeping intent visible in repository
Append-only JSONL trace	Immutable, crash-safe, easily inspectable via `grep` or `jq`	Addresses **Trust Debt** via auditability
SHA-256 content hash (not file mtime)	Filesystem timestamps are unreliable; content hash ensures correctness	Prevents **Context Rot**
Pre-hook fail-fast chain	Stops execution immediately on violation; prevents partial writes	Minimizes **Trust Debt**
Structured JSON errors to LLM	Enables autonomous recovery and retry	Reduces **Cognitive Debt**
Fail-open on hook crash	Prevents governance bugs from blocking development	Trade-off between safety and productivity

Module-level caches	Simplicity and performance; reset on restart	Acceptable trade-off for v1
Hash from disk, not tool payload	Guarantees trace reflects real filesystem state	Eliminates forensic ambiguity

---

6. Trade-offs, Limitations, and Deviations

Known Limitations

Limitation	Impact	Mitigation
Scope enforcement for `apply_patch`	Embedded file paths may not be extracted reliably	Authorization layer + LLM scope awareness
Module-level cache persistence	Cache survives hot reload	Document restart requirement
Single-workspace assumption	No explicit multi-root support	Extendable in future
No cross-agent intent locking	Parallel agents may share intent	Future lock-file mechanism

---

Deviations from Original Plan

Original Plan	Actual Implementation	Reason
Tools throw on failure	Wrapped `pushToolResult` instead	`ExecuteCommandTool` does not throw
`CLAUDE.md` inside `.orchestration/`	Placed at workspace root	Anthropic convention
Scope violation always dialog-based	LLM often self-corrects before hook fires	Intent injection improved behavior

---

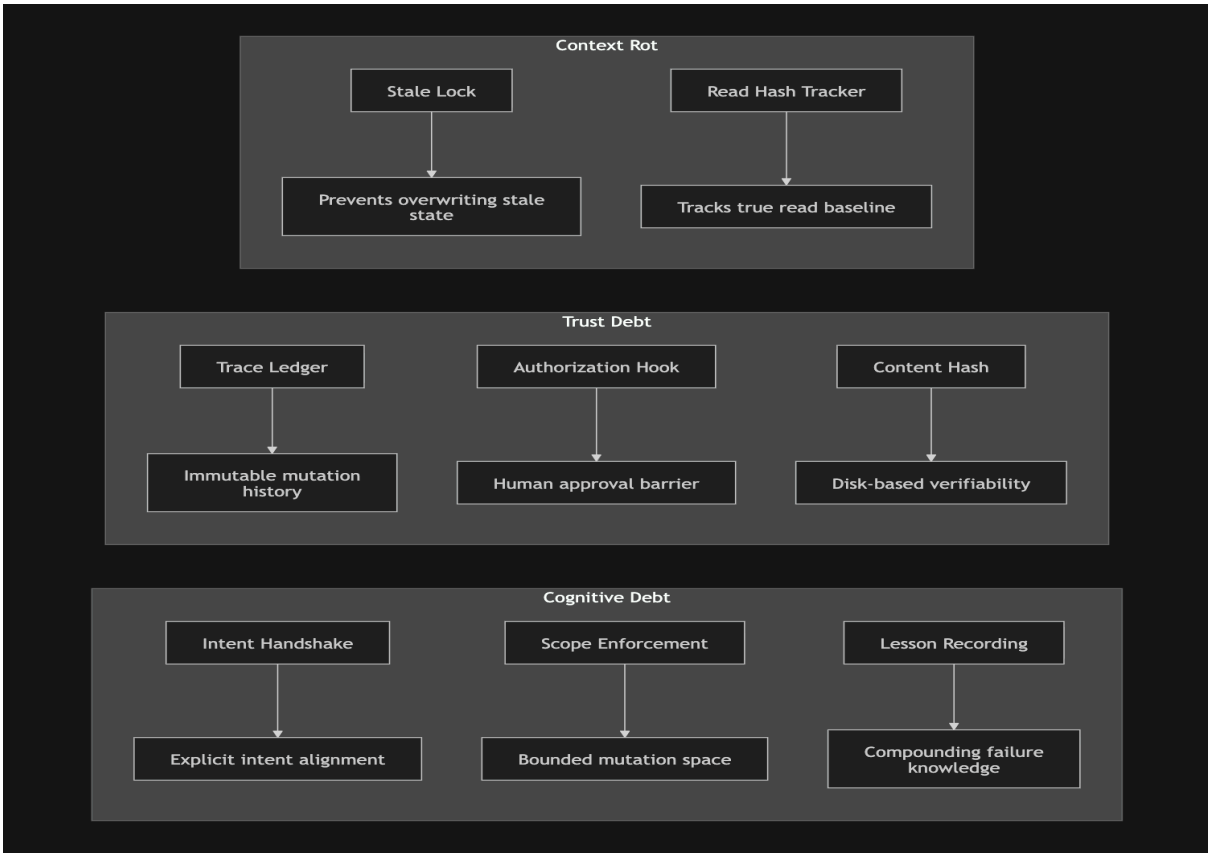
7. Achievement Summary & Reflective Analysis

7.1 Honest Inventory — What Was Built

Component	Status	Evidence

Intent Handshake	✅ Fully implemented	Enforced before destructive tools
Scope Enforcement	✅ Fully implemented	Minimatch + traversal prevention
Authorization Gate	✅ Fully implemented	VS Code modal
Trace Ledger	✅ Fully implemented	Immutable JSONL entries
Mutation Classifier	✅ Fully implemented	Deterministic ledger-based classification
Stale Lock	✅ Fully implemented	Verified stale detection
Lesson Recorder	✅ Fully implemented	CLAUDE.md append + dedup
`apply_patch` path parsing	⚠️ Partial	Embedded path extraction limited
Cross-agent locking	❌ Not implemented	Out of v1 scope
Multi-root support	❌ Not implemented	Future enhancement

7.2 Debt Mapping — How Each Component Repays Technical Debt



7.3 Technical Lessons Learned

1. **Tool error handling is inconsistent across Roo-Code tools.** Some tools do not throw on failure.
2. **System prompt reconstruction requires idempotent injection.**
3. **Module-level state persists across hot reloads.**
4. **LLM proactive scope awareness improves enforcement robustness.**
5. **Disk-based hashing is mandatory for forensic accuracy.**

7.4 Concrete Next Steps

Priority	Task	Effort	Debt Addressed
High	Extract file paths from `apply_patch` content	2h	Cognitive Debt
High	Reset classifier cache on workspace reload	30m	Context Rot
Medium	Intent reservation lock file	4h	Trust Debt
Medium	Trace + lesson UI notifications	2h	Trust Debt

Low	Multi-root workspace support	4h	—
Low	Governance dashboard webview	8h	Cognitive Debt

---

8. Test Results

```
✓ src/hooks/__tests__/HookEngine.spec.ts     (Phase 2)
✓ src/hooks/__tests__/phase3.spec.ts         (Phase 3)
✓ src/hooks/__tests__/phase4.spec.ts         (Phase 4)

Test Files  3 passed (3)
     Tests  49 passed (49)
  Duration  < 2s
```

9. Conclusion

The AI-Native Git Layer demonstrates that AI-assisted development can be governed through deterministic enforcement rather than informal trust.

This implementation proves that:

- Intent alignment can be made explicit and enforceable.
- Destructive actions can be bounded by scope and human approval.
- File mutations can be cryptographically verifiable through disk-based hashing.
- Mutation history can be ledger-derived and restart-consistent.
- Concurrency risks can be mitigated via optimistic locking.

By combining intent declaration, scope enforcement, authorization gating, immutable trace serialization, and stale state detection, this system reduces Cognitive Debt, Trust Debt, and Context Rot in AI-native workflows.

While certain enhancements remain (cross-agent locking, multi-root workspace support, deeper `apply_patch` parsing), the current architecture establishes a deterministic governance foundation suitable for production-grade AI development tooling.

This project demonstrates that AI agents do not require blind trust — they require structured constraints, observable state, and enforceable contracts.