How Dataloop Safely Improves Itself: A 60-Day Study of Governed AI Self-Improvement in Datakult 2.0

Philippe Lobo Kung
Clinic Of AI

19 June 2026

Abstract

This paper does not claim novelty in self-modifying coding agents, automated agent design, evolutionary code search, or cognitive architectures individually. Its contribution is their operational synthesis into a continuously-running, trace-driven, gate-mediated recursive self-improvement (RSI) layer inside a production cognitive stack. Datakult 2.0 treats RSI as an immune-system layer: runtime pathology becomes structured trace evidence; trace evidence becomes candidate repairs; candidate repairs pass through deterministic promotion gates; accepted repairs modify the substrate that emits future traces. The system separates creativity, supplied by stochastic LLM workers, from authority, supplied by deterministic gates and an append-only audit chain.

This paper reports a 60-day telemetry window from a single private production codebase estimated at 750-850K Python LOC. From 2026-03-23 through 2026-05-21, the current first-parent `origin/main` history records 4,347 commits. A reproducible conservative prefix classifier finds 1,794 structural RSI/shepherd commits (41.3%); a broader internal audit chain records 4,267 promotion verdicts, including 1,915 accepted and 1,490 rejected outcomes. The two-week structural-prefix trajectory was 10.7, 2.7, 37.1, 48.9, and 61.2 commits/day; the final six-day bucket remains suggestive rather than statistically decisive. This paper introduces Operator Intervention Half-Life as a candidate metric: the time required for a repeated human-intervention class to become detected, patched, gated, and retired. The counts are evidence, not the centerpiece; the centerpiece is an engineering pattern for governed self-improvement under bounded authority, observable failures, and audit memory.

Contents

Positioning in one sentence. This paper does not claim novelty in self-modifying coding agents, multi-agent systems, cognitive architectures, or automated evaluators individually. Its contribution is the operational synthesis of these elements into a continuously-running, trace-driven, gate-mediated RSI layer inside a production cognitive stack — *governed self-improvement as an operating-system pattern*. The closest published analogues are Schmidhuber's Gödel Machine (2003), Soar / ACT-R / OpenCog Hyperon as cognitive-architecture lineage, Darwin Gödel Machine (Zhang et al. 2025), Live-SWE-agent (Xia et al. 2025), ADAS / Meta Agent Search (Hu et al. 2024), and AlphaEvolve (Novikov et al. 2025). This work sits at a different *operating point* in the same design space: a single private production codebase, continuous operation, deterministic promotion gates as the authority boundary, an append- only audit chain as memory, and human operator intervention treated as a measurable, retirable signal rather than a permanent budget line.

AI-use disclosure. Claude Opus 4.7 (Anthropic), OpenAI Codex App Server, and related LLM tools were used for drafting, code review, operational analysis, and revision support. The human author reviewed the manuscript and accepts responsibility for all claims, citations, data, and conclusions.

Lay Summary

Plain-language version for readers outside the AI-systems community.

This paper is *not* about being the first system whose AI edits its own code. That story is already taken — by the Darwin Gödel Machine, by Live-SWE-agent, by AlphaEvolve, by the ADAS / Meta Agent Search line of work, and historically by the cognitive-architecture lineage of Soar, ACT-R, and OpenCog. What this paper is about is what happens when you try to turn that idea into something that *runs every day on a real codebase as part of normal operations*, not as a benchmark experiment.

Concretely: I built and operated a five-system cognitive stack ("Datakult 2.0") in which one of the systems — the recursive self- improvement layer — behaves like an **immune system** for the rest of the stack. It watches what is broken or suboptimal, clusters those observations into structured trace evidence, proposes candidate repairs by dispatching large-language- model workers, decides whether to accept those repairs using a *deterministic* promotion gate (not an LLM), records everything in an append-only audit chain, and then the next cycle observes the substrate the previous cycle just modified. The LLM is allowed to be *creative*; it is not allowed to be *authoritative*. Authority lives in the gate and the audit chain. This work treats Codex-style coding agents as creative workers inside a larger operating harness; the safety, retry, state, audit, and promotion semantics are supplied by Dataloop's outer system rather than by the worker model itself.

I operated this system across a 60-day window. The current first-parent `origin/main` history records **4,347** commits in that window. A reproducible conservative prefix classifier counts **1,794 structural RSI/shepherd commits (41.3%)**; a broader internal audit chain records **4,267** promotion verdicts through the same window, including 1,915 accepted and 1,490 rejected outcomes.

The most recent six-day bucket reached ~**60 structural RSI commits per day**, after an early low single-digit bucket and a later higher sustained operating regime. The gate accepts 45% of candidates, rejects 35%, and skips/recovers/holds the rest. One recent incident illustrates the trajectory: the human operator tried to manually fix a problem and discovered the system had shipped an equivalent structural repair thirty minutes earlier.

These numbers are evidence, not the centerpiece. The centerpiece is the *engineering pattern*: a continuously operating cognitive stack in which self-improvement is treated as a governed subsystem, with sensors, memory, audit, gates, self-repair, and operator-loop automation. This paper introduces a new candidate metric — **Operator Intervention Half-Life** — defined as the time required for a repeated human-intervention class to be detected, patched, gated, and retired by the loop. That metric reframes progress toward autonomy as *reduction in human-interrupt burden*, not as benchmark score.

The dominant public story of AI progress has been foundation- model scaling. A parallel research direction has been quietly emerging around agentic systems, self-improving scaffolds, automated agent design, and AI-driven scientific discovery. Dataloop sits in that second direction, but pushes it toward *production operations* rather than benchmark-only improvement. The thesis this paper offers is small and concrete: a plausible route to artificial general intelligence is unlikely to consist of a naked model alone. It may require a persistent cognitive operating system that combines memory, perception, planning, reflection, tool use, error recovery, social feedback, and self-modification *under stable governance*. This paper is one production case study of what that governance might look like.

The system runs on standard infrastructure (an LLM, a Python codebase, a git repository, a Linux server with a GPU). The novelty is in the *composition*: trace-driven sensors, a deterministic promotion gate, an append-only audit chain, and a self-healing supervisor. Removing any one of these causes documented regressions, as discussed in ğ9.6.

What might generalize: the architectural patterns described here are language- and provider-agnostic. This paper treats it as plausible that any sufficiently structured codebase could host an analogous loop. What probably doesn't generalize: the specific velocity numbers reported here depend on the codebase, the model, and the operator's training data. The shape of the curve — slow start, infrastructure inflection, sustained acceleration — is what this paper predicts may replicate.

---

1. Introduction

The hypothesis that artificial agents could improve themselves recursively dates at least to Good (1965), who imagined an "ultraintelligent machine" whose ability to improve its own design would launch an intelligence explosion. Sixty years later, the engineering realization of that idea remains elusive. Most reported attempts collapse into one of two failure modes: (1) the agent proposes changes it cannot validate, and external operators end up acting as the validator, or (2) the agent's loop wedges, dies, or silently degrades, requiring an operator to restart it. In both modes the human remains the rate-limiting step.

This paper documents an alternative: an *engineered* RSI loop in which the human operator's role has empirically reduced to verification, and in which the loop's failure-modes are themselves

detected, repaired, and prevented from recurring by the same loop. This paper does not claim to have built an "ultraintelligent" system; it claims only that the bounded loop works within the operating regime described here, and that the sixty-day operational record is interesting enough to share.

The central hypothesis of this work is that continuous learning is not a property of a naked model alone. It is a property of a governed substrate: memory, sensors, evaluators, recovery mechanisms, authority boundaries, and feedback loops arranged so that experience can safely modify future behavior. Datakult 2.0 is not presented as AGI. It is presented as an early engineering attempt to build the kind of substrate around contemporary language models within which continuous self-improvement could occur.

1.0 What This Paper Is Not

Before stating contributions, this paper explicitly enumerates the things this paper does *not* claim, because each of them is already occupied by prior or contemporaneous work and a careless reading of the present system might suggest otherwise:

1. **Not the first self-modifying coding agent.** The Darwin Gödel Machine (Zhang et al. 2025, arXiv:2505.22954) describes a self-improving coding agent that modifies its own code and validates changes empirically on coding benchmarks (SWE-bench 20.0% → 50.0%; Polyglot 14.2% → 30.7%). Live-SWE-agent (Xia et al. 2025, arXiv:2511.13646) reports a runtime self-evolving software agent that modifies its own scaffold while solving software problems, and positions itself explicitly against DGM.
2. **Not the first automated agent designer.** ADAS / Meta Agent Search (Hu et al. 2024, arXiv:2408.08435) argues that agents can be defined in code and automatically improved by a meta- agent that invents new agent designs — covering much of the "agents improving agent architecture" angle.
3. **Not the first evaluator-driven code evolution system.** AlphaEvolve (Novikov et al. 2025) combines a large language model with automated evaluators to evolve algorithms via direct code changes, including practical infrastructure optimization and mathematical discovery.
4. **Not the first complex cognitive architecture.** Soar (Laird 2012), ACT-R (Anderson 1996), and OpenCog / Hyperon (Goertzel et al. 2014) are all multi-component cognitive-architecture research programs in this exact tradition. The five-system structure of Dataloop is in dialogue with that lineage, not independent of it.

What this paper *does* claim is the operational synthesis. None of the ingredients are new in isolation; the combination — a single private production codebase with trace-driven sensors, deterministic promotion gates, append-only audit chain, self- healing supervisor, machine-operable PRD lifecycle, and operator-loop automation, running continuously for sixty days — is, to our knowledge, undescribed in the published literature. Section 2 discusses each of these analogues at length; Section 2.3 states the specific composition novelty.

1.1 Contributions

This paper makes the following contributions, framed as *operational synthesis* rather than as invention of any single ingredient:

1. **An immune-system framing for RSI.** This paper argues that prior systems treat self-improvement primarily as benchmark performance optimization. This paper treats RSI as an *immune-system layer* in a continuously operating cognitive stack: the loop observes runtime pathology, repairs the substrate, hardens the gate, and reduces future operator intervention. This paper documents the engineering decisions that follow from that framing (ğ4, ğ8.1, ğ8.6).
2. **A creativity-vs-authority separation.** This paper documents an explicit architectural separation in which LLM workers are *creative and stochastic*, the promotion gate is *deterministic and authoritative*, the audit chain is *append-only and load-bearing for memory*, and the shepherd skill is *human verification automated as a measurable target* (ğ4, ğ8.2). This paper argues that this assignment of creativity and authority to different components is the load-bearing safety property of the system.
3. **Traces as curriculum.** This paper argues that the curriculum for self-improvement need not be human-labeled tasks; it can be the system's own operational scar tissue, provided that scar tissue is converted into structured trace evidence and gated repairs. This paper documents how **31 registered harvester layers** in `Datakult_AGI_Utils/rsi/trace_exporter.py` and a **56-member `EventType` enum** in `Datakult_AGI_Utils/rsi/trace_schema.py` produce a rolling, self-renewing curriculum (ğ5, ğ8.4). Not every enum member is emitted in every 24-hour harvest window, so this paper reports the registry count and enum size separately rather than conflating source layers with event classes.
4. **PRDs as machine-operable product substrate.** This paper documents a Phase 2 pipeline in which product-requirements documents are first-class machine-readable artifacts, walked through a lifecycle (`proposed` → `approved` → `implementing` → `implemented`) by deterministic primitives. This pushes self-improvement beyond "agents writing code" toward what this paper calls *autonomous software-organization design* (ğ4.3, ğ8.5).
5. **Operator Intervention Half-Life (OIH).** This paper introduces a candidate central metric: the time required for a repeated human-intervention class to become detected, patched, gated, and retired by the RSI loop. This paper argues that OIH is a better north-star for governed self-improvement than benchmark score, because it directly measures progress toward operator-free operation (ğ6.7).
6. **Operational data.** This paper presents 60 days of commit-history telemetry from a continuously-running supervised RSI campaign, including velocity trajectory, quality scorecards, and incident-recovery traces. This paper treats this data as *evidence for the pattern*, not the centerpiece of the paper.
7. **Failure analysis.** This paper documents four classes of failure the loop has had to learn to detect and self-repair: dual- lifecycle directory pollution, worktree-commit loss during promotion, backend rate-limit wedges, and silent stall conditions.
8. **A path to full autonomy.** This paper names the three engineering milestones that appear to separate the current state from "operator-free" operation, and provide a calibrated estimate of their timeline based on observed velocity.

1.2 Non-goals

This paper does *not* claim that the system has achieved artificial general intelligence, that it can improve its own fundamental architecture, or that it has solved alignment. RSI here is bounded: it operates within an editable-paths allowlist, must pass locked regression tests, and is subject to a promotion gate that is itself an audited artifact. The system improves *itself within those bounds*. The bounds are set by humans, and the questions of where to set them remain firmly outside the

loop.

1.3 First Principles: The Conceptual Model

For readers approaching this work from a theoretical rather than engineering angle, the system can be understood as a closed control loop with five conceptual primitives:

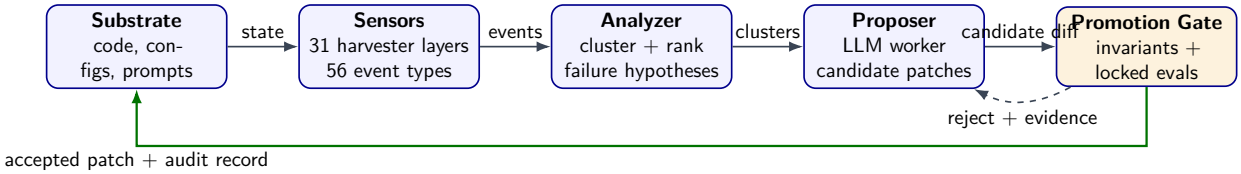

Figure 0: The five-primitive conceptual model. Substrate emits state; sensors structure that state; analyzer prioritizes what to address; proposer generates a candidate; evaluator gates promotion. The loop closes when accepted patches modify the substrate, which then emits new state in the next cycle.

This decomposition makes three claims explicit:

1. **Self-improvement is a control problem.** The substrate is the *plant*; the loop is the *controller*. Stability, observability, and controllability — the classical control properties — all have direct analogues here. The promotion gate is the *safety filter*.
2. **The five primitives are independently substitutable.** Any sensor can be replaced; any analyzer policy can be swapped; any proposer model can be exchanged; the evaluator's invariants can evolve. The primitives are *interfaces*, not implementations.
3. **The loop is reflexive when the substrate includes itself.** When the substrate is the code that implements the loop, the loop modifies its own machinery. This is what makes the system *recursively* self-improving rather than just self-improving. Most prior systems (e.g., AlphaGo) modify their *parameters*; this system modifies its *source code*, including the source code of the loop primitives themselves (gate excepted).

The rest of the paper instantiates each primitive concretely and reports on what happens when the loop runs across the 60-day window described in ğ6 — with supervised restarts and the explicit interventions catalogued in ğ7.4.

1.4 Biological Priors and Governed Complexity

Biological learning does not begin from a blank substrate. A newborn organism arrives with morphology, reflexes, sensorimotor circuits, drives, homeostatic regulation, and architectural priors shaped before its first experience. Recent work on neural-circuit architectural priors makes this concrete: animals are born with nervous-system structure shaped by evolution, and a horse can walk within hours of birth while rapidly improving with practice (Bhattasali et al. 2024). The analogy to artificial systems is imperfect but useful. An AI system expected to learn continuously from experience may likewise require more than a model and a prompt; it may require a governed substrate - memory, sensors, evaluators, recovery mechanisms, audit, authority boundaries, and feedback loops - within which experience can safely modify future behavior.

This paper does not attribute the biological-priors framing to Karpathy directly. Karpathy is cited for the training-recipe and agentic-coding influences described below; the horse analogy is cited to the neural-circuit architectural-priors literature. The narrower engineering claim is that governed complexity is not ornamentation: it is the substrate that makes continuous learning auditable, recoverable, and safe enough to operate.

---

2. Related Work

Self-improving systems. Schmidhuber's Gödel machine (2003) proved that a self-rewriting program can be safe under formally-provable rewrites; in practice the proof-search dominates the cost. This paper takes an engineering rather than formal route: candidate changes are validated empirically by locked regression tests, not provably.

Self-play and learned optimization. AlphaGo (Silver et al., 2016) and AlphaGo Zero (Silver et al., 2017) demonstrated that self-play with a stable evaluator can produce superhuman policy through iterated improvement. RSI here is structurally similar: the locked evaluation harness is the stable evaluator, and the LLM-worker is the policy proposer. The key difference is that the policy here is source code, not move probabilities.

Recursive code improvement. The recent "Ralph loop" (Huntley, 2025) and Anthropic's *Building Effective Agents* (Anthropic, 2024) describe single-agent and orchestrated agent patterns that drive LLM-mediated code change. The present system extends that pattern with: (1) explicit phase separation, (2) a promotion-audit chain, (3) a self-healing supervisor, and (4) trace-driven candidate selection.

Darwin Gödel Machine (DGM). The closest *published* analogue to the present work is the Darwin Gödel Machine (Zhang et al. 2025, arXiv:2505.22954). DGM maintains a population of coding agents, mutates their tooling and prompts, evaluates candidates on SWE-bench / Polyglot, and keeps an empirical archive of improvements (SWE-bench 20.0% $\rightarrow$ 50.0%; Polyglot 14.2% $\rightarrow$ 30.7%). Where DGM operates as a benchmark-driven population evolutionary system on a public coding-task substrate, Datakult 2.0 RSI is a single-agent control loop operating on a single private production codebase estimated at 750-850K Python LOC with a deterministic composite-score gate, an append- only audit chain, and live main-branch commit history. This paper views DGM and Dataloop as complementary points in the design space: DGM optimizes for *what an agent can solve on a benchmark*; Dataloop optimizes for *what a single codebase will accept as a safe self-modification at production cadence*. Section 9.1 discusses what each framework might learn from the other.

Live-SWE-agent. Xia et al. (2025, arXiv:2511.13646) describe a runtime self-evolving software agent that modifies its own scaffold while solving software problems. Live-SWE- agent explicitly positions itself in the lineage that includes DGM and other self-improving software-agent work, and is contemporaneous with the present paper. The structural overlap with Dataloop is significant: both modify the agent scaffold at runtime; both evaluate against a stable substrate; both treat self-modification as a first-class operation. The operating-point difference is that Live-SWE-agent appears to target per-problem self-evolution within the SWE-agent ecosystem, while Dataloop targets a single continuously-running loop on a single private codebase with a deterministic gate and append-only audit chain. Neither result appears to subsume the other; they are different operating points in the same design space.

Automated Design of Agentic Systems (ADAS). Hu et al. (2024, arXiv:2408.08435) introduce *Meta Agent Search*: a meta-agent that programmatically invents and refines agent designs, with new agents defined in code and evaluated on held-out tasks. ADAS covers much of the "agents improving the architecture of agents" angle. The structural overlap with Dataloop is in the *meta* layer: Dataloop's Phase 2 feature pipeline can be read as a constrained form of agent-design automation, in which the search is over PRD-shaped design artifacts and the evaluator is the deterministic promotion gate rather than a held-out task suite. ADAS optimizes for *best agent design under a benchmark*; Dataloop optimizes for *what an existing production cognitive stack will accept as a self-modification under a fixed gate*. The two systems share the conviction that agent architecture is a first-class optimization target.

AlphaEvolve. Novikov et al. (2025, Google DeepMind) describe a coding agent for scientific and algorithmic discovery that combines large language models with automated evaluators to evolve better algorithms by making direct code changes, with reported wins on mathematical discovery and production infrastructure optimization (Borg scheduling, matrix multiplication kernels). AlphaEvolve is the cleanest published example of *evaluator- driven code evolution at scale*. The structural overlap with Dataloop is at the loop level: a creative LLM proposer plus a deterministic evaluator plus an archive of accepted changes. The operating-point difference is that AlphaEvolve targets *algorithmic search* on bounded problems, while Dataloop targets *continuous maintenance and self-modification* on an entire multi-module production codebase. This paper reads AlphaEvolve as the strongest argument that the pattern of "creative model + deterministic evaluator + archive" generalizes; Dataloop is one attempt to instantiate that pattern as an *operating-system layer* rather than as a research artifact.

Cognitive architectures (Soar, ACT-R, OpenCog / Hyperon). The five-system structure of Dataloop sits in dialogue with the long-running cognitive-architecture research program. Soar (Laird 2012) and ACT-R (Anderson 1996) are explicitly multi- component cognitive architectures aimed at general intelligent behaviour, with chunking, declarative/procedural memory separation, and unified theories of cognition. OpenCog / Hyperon (Goertzel et al. 2014; Hyperon 2023) is explicitly framed as a multi-paradigm cognitive architecture oriented toward AGI, combining symbolic, probabilistic, and neural substrates. This paper does not claim that Dataloop's five-system decomposition is a successor or refutation of these systems; this paper reads Dataloop as a *production-systems-engineering instance* of the same intellectual tradition, biased toward Linux processes, git history, and Redis pub/sub as the substrate rather than toward unified theoretical formalisms. The contribution is operational, not theoretical.

Continual / lifelong learning. Parisi et al. (2019) survey continual learning under catastrophic forgetting; Dataloop's locked-eval gate is a discrete analogue, preventing the loop from regressing previously-fixed behaviors by requiring all locked tests to continue passing on every promotion.

Karpathy's training recipe and agentic-coding frame. Karpathy (2019) describes a neural-network training "recipe" emphasizing incremental, verifiable improvement with strict before/after measurement. The Phase 3 sub-loop is named "Karpathy-Mode" for two reasons: the 2019 recipe's emphasis on incremental, measured, before/after improvement, and Karpathy's later public framing of vibe coding / agentic coding as a natural-language software-development interface. In Dataloop, that inspiration becomes a bounded calibration loop: one file, one metric, one time budget, and one deterministic gate. This paper does not attribute the biological-priors argument to Karpathy; it uses Karpathy for the measurement discipline and the agentic-coding workflow influence.

Software engineering automation. SWE-bench (Jimenez et al., 2024) demonstrated that LLM

agents can solve real GitHub issues; the present system extends this to a *continuously-running* production loop on a single codebase with strict promotion gates, and reports a recent six-day peak velocity on the order of 60 structural RSI commits per day, with lower sustained rates over earlier two-week buckets.

2.1 Comparison with Prior Systems

System	Operating point	Self-modifies agent code?	Evaluator / authority boundary	Scope
Soar / ACT-R / OpenCog Hyperon	Cognitive-architecture research programs	No autonomous source-code self-modification	Theoretical and experimental validation	Synthetic and lab tasks
AutoGPT / Aider / AutoGen / ChatDev / Devin	Task, session, or per-project agent systems	None publicly reported as persistent self-modification	User review, local tests, or internal phase checks	Arbitrary tasks or single repos
SWE-agent / SWE-bench	Benchmark software-engineering agent and harness	No	SWE-bench evaluation harness	Per GitHub issue
Voyager	Persistent skill acquisition in Minecraft	Skill-library growth, not source self-modification	Game-internal feedback	Synthetic environment
OpenAI o1	Inference-time deliberation	No persistent self-modification reported	Single-response evaluation	N/A
ADAS / Meta Agent Search	Meta-agent invents agent designs	Yes	Held-out task benchmarks	Coding and reasoning benchmarks
AlphaEvolve	LLM proposer plus automated evaluator	Yes, candidate algorithms	Automated problem-specific evaluators	Algorithmic discovery and infrastructure optimization
Darwin Gödel Machine	Population-evolutionary coding-agent loop	Yes, agent tooling and prompts	Benchmark fitness on SWE-bench / Polyglot	Public coding benchmarks
Live-SWE-agent	Runtime self-evolving software-engineering agent	Yes, own scaffold	SWE problem-solving outcomes	SWE-agent ecosystem
Datakult 2.0 RSI	Continuous gate-mediated self-modification on one production codebase	Yes, gate-mediated RSI machinery edits	7-term composite scorer, hard gates, locked evals, and append-only audit chain	Private repo estimated at 750-850K Python LOC plus 34K-LOC sibling

Table 0: Comparison with related systems. Self-modification of agent code is **not** unique to Dataloop: ADAS, AlphaEvolve, DGM, and Live-SWE-agent all describe agents that modify

their own code. What differs is the *operating point*. ADAS optimizes agent design under a held-out benchmark; AlphaEvolve evolves algorithms with an automated per-problem evaluator; DGM runs a population-evolutionary loop on SWE-bench / Polyglot; Live-SWE-agent self-evolves per software-engineering problem within the SWE-agent ecosystem. Dataloop is a single-agent control loop that runs continuously on one production codebase, with a deterministic composite-score gate as the authority boundary and an append-only audit chain as memory. This paper reads all of these systems as complementary points in a shared design space; none of them subsumes any of the others.

2.2 Origin and Evolution of Dataloop RSI

The system described in this paper did not begin as a code-modifying loop. Its genesis is dated **2026-03-17**, when the project's first foundational PRD was authored by the OpenAI Codex agent under the title *Mem0 Memory Locker & Recursive Self-Improvement Fabric* (v1.0.0). That document still exists verbatim at `Future_Updates/mem0_memory_locker_recursive_self_improvement/` and is, to our knowledge, the earliest written description of what would become Dataloop's System Four.

The original framing was emphatically not "an LLM that rewrites code." It was "a memory system that learns from outcomes." Specifically, the PRD asked for:

- An explicit **memory taxonomy** with kinds such as `identity_profile`, `user_preference`, `task_outcome`, `procedural_tool_use`, `reasoning_outcome`, `reward_signal`, `self_reflection`, `belief_update`, `corruption_report`, and `fold_summary`.
- A **mutation decision engine** supporting ADD, UPDATE, MERGE, QUARANTINE, DELETE, and NONE — with evidence, confidence, and human-readable reason on every mutation.
- A **reward ledger** linking positive and negative outcomes to the memories they reinforced or contradicted.
- A **maintenance loop** that would consolidate duplicates, review contradictions, and clean stale records on a bounded schedule.
- **Locker observability** so operators could inspect mutation history, quarantine state, reward totals, and retrieval quality.

Under this framing, recursive self-improvement was *recursive learning of memory and beliefs* — Dataloop getting better by revising what she remembered, not by rewriting her source code.

2.2.1 The Inflection: From Memory Governance to Code Modification

Between March and late April 2026, several converging infrastructure threads turned the memory-governance vision into a code-modifying loop:

1. **Trace-harvester coverage matured.** During the reporting window the system reached **31 registered harvester layers**. The operational subsystems are materialized by the `_HARVESTER_REGISTRY` tuple in `Datakult_AGI_Utils/rsi/trace_exporter.py`; the current `EventType` enum in `Datakult_AGI_Utils/rsi/trace_schema.py` contains **56 members**. Not every enum member is emitted in every 24-hour harvest window, so this paper distinguishes the harvester-layer count from the enum-size count. Earlier in development both numbers were smaller; each added layer or event class unlocked a new class of detectable failure. **Reproducibility note:** Appendix D ğD.2 gives the exact commands that print the

current harvester count and full `EventType` enum size on any clone.

2. **The cluster-and-dispatch primitive emerged.** Once structured traces were available across enough layers, failure clustering became feasible, and the natural next step was to dispatch an LLM worker into a worktree to address each cluster.
3. **The supervised continuous runner** (`rsi_supervise.sh`) came online, allowing campaigns to roll automatically and compounding learning across iterations.
4. **Phase 2 — Feature Pipeline (local 2026-04-21 / UTC 2026-04-22):** the autonomous Feature Implementation Pipeline was added, scanning `Future_Updates/approved/` PRDs and dispatching workers to implement them with strict invariant checks. This is the point at which RSI became a code-modifying agent at scale. (The date appears as 2026-04-21 in the local-timezone commit log and 2026-04-22 in UTC-formatted artifacts; these refer to the same event.)
5. **Phase 3 — Karpathy-Mode (local 2026-04-21 / UTC 2026-04-22):** the Karpathy-mode sub-loop was added the same day, running bounded one-file / one-metric / one-time-budget experiments with delta-from-baseline regression checks. This formalized parameter optimization as a peer of bug-fix and feature dispatch. **Caveat:** the lane infrastructure and Karpathy Triplet specification are in place, but most of the 24 declared lanes did not empirically fire under load during the reporting window; see ğ8.9 ("Karpathy-mode has been mostly dormant") for the honest dormancy analysis and ğ11 for the calibration roadmap.
6. **Codex App Server worker-backend integration (2026-05-18):** RSI transitioned to using OpenAI's Codex App Server interface (Dataloop's internal worker-backend module is named `codex_app_server`) as the primary worker-dispatch path. The repo-scoped Codex/OpenAI configuration for this path uses `gpt-5.5` with `xhigh` reasoning; it is not the local `gpt-oss-120b` vLLM runtime. The `/shepherd` operator skill uses the same high-capability `gpt-5.5` / `xhigh` route. Local `gpt-oss-120b` is reserved for judge and analysis roles. Claude remains available in the outer failover/backend chain, while app-server internal automatic fallback is disabled by default in the current `config/rsi/codex_app_server.yaml`. The original Codex-exec path remains as a manual rollback. Dataloop supplies its own production safety, retry semantics, state handling, and promotion logic around the Codex App Server interface (which OpenAI documents as experimental and primarily intended for local development); none of those production guarantees are assumed to come from the Codex App Server itself.

The memory-governance vision from the March 17 PRD has not been abandoned. AITP still uses Mem0 as a memory locker, the mutation primitives still exist in `Mem0_Master_Manager_Client.py`, and reward signals still feed back into retrieval. What happened is that *code-improvement was added as a sibling capability of memory-improvement*, and the code-improvement capability has dominated subsequent engineering attention because it produces more visible artifacts (commits).

2.2.2 Inspirational Sources

Tracing the project's intellectual provenance — both as documented in the original PRD and as observable in subsequent naming, comments, and skill specifications — the following sources directly informed the design:

Source	Influence on Dataloop RSI

Mem0 official documentation (graph memory, advanced retrieval, metadata filtering, update/delete operations, feedback API, webhooks)	The original PRD cites Mem0 docs directly. The locker taxonomy, mutation primitives, and reward signal pattern come from this lineage.
Karpathy (2019; 2025-2026), training recipe and agentic coding	The Phase 3 sub-loop is named "Karpathy-Mode" for two reasons: Karpathy's 2019 recipe emphasizes incremental, verified, before/after-measured improvement, and his later public framing of vibe coding / agentic coding reframes software development as natural-language orchestration of coding agents. In Dataloop, this becomes a bounded calibration loop: one file, one metric, one time budget, one deterministic gate.
Schmidhuber (2003), *Gödel machines*	The intellectual root for "a program that rewrites itself under safety constraints." This work replaces Gödel's proof obligation with locked-eval empirical validation, but the spirit is preserved.
Silver et al. (2016, 2017), *AlphaGo / AlphaGo Zero*	The structural analogy of "stable evaluator + improving proposer" is the AlphaGo pattern. Our locked-eval gate is the stable evaluator; the LLM worker is the proposer.
Huntley (2025), *Ralph Wiggum as a software engineer* (the Ralph Loop pattern)	The original continuous-mode runner draws directly from Ralph-loop semantics. Earlier RSI skill bug-fix entries explicitly reference "Ralph-era behavior" being superseded by the current Codex App Server outer runner.
SWE-bench (Jimenez et al., 2024) and SWE-agent (Yang et al., 2024)	The benchmark-style "agent fixes a real issue" pattern, generalized to "agent runs continuously fixing whatever cluster emerges from traces."
Voyager (Wang et al., 2023)	The skill-library persistence pattern. Our equivalent is the rolling audit chain plus the Phase 2 PRD lifecycle: knowledge accumulates as a *commit history*, not as a learned skill set, but the principle of "make persistent what you discover" is shared.
Pike (1989), *Notes on Programming in C*, Rule 5: "Data dominates"	The repeated lesson from this project. Most of the system's most-impactful improvements have been changes to how state is represented (the audit chain, the heartbeat, the PRD lifecycle), not changes to the algorithms operating on the state.
Sutton (2019), *The Bitter Lesson*	The decision to lean on LLM workers rather than hand-crafted patches. This work accepts the bitter lesson at the worker level (LLMs propose) while imposing structure at the gate level (deterministic invariants and locked tests).

The synthesis is original to this project. To our knowledge, no prior published system combines (a) a memory-locker framing inherited from the Mem0 lineage, (b) a Karpathy-style parameter-tuning sub-loop, (c) a SWE-agent-style cluster dispatcher, (d) an AlphaGo-style stable evaluator, and (e) an append-only audit chain into a single continuously-operating loop.

2.3 Novelty Positioning

The manuscript owes the reader a precise statement of what this paper claims and what it does not. The novelty claim runs as follows:

> *This work does not claim novelty in self-modifying coding agents, multi-agent systems, cognitive architectures, or automated evaluators individually. Its contribution is the operational synthesis of these elements into a continuously running, trace-driven, gate-mediated RSI layer inside a production cognitive stack. The system treats self-improvement as an immune-system function: runtime pathology becomes structured trace evidence; trace evidence becomes candidate repairs; candidate repairs pass through deterministic promotion gates; accepted repairs modify the substrate that emits future traces. This paper argues that this closed loop is a practical engineering pattern for governed self-improvement in complex AI systems.*

This is the central claim of the paper. The remainder of ğ2.3 unpacks the four operational claims that follow from it.

2.3.1 RSI as Immune System, Not Benchmark Optimizer

Prior published self-improving-agent systems are framed primarily as *benchmark optimizers*: DGM optimizes SWE-bench and Polyglot scores; ADAS optimizes held-out reasoning-task performance; AlphaEvolve optimizes algorithmic objective functions; Live-SWE-agent optimizes per-problem solving rates. Dataloop is framed as a **system-maintenance immune layer** inside a living codebase: harvest traces, detect failure patterns, cluster them, generate repairs, gate them, commit them, then watch the new system emit new traces. The distinction is not cosmetic. A benchmark-optimizer has a fixed objective and a finite improvement budget; an immune system has an *open-ended* objective ("keep the organism functional in the face of novel failures") and a *continuously refreshing* adversary surface (the next failure has not yet occurred). The engineering decisions follow accordingly: trace harvesters that admit novel `EventType`s, gates that admit new invariants, PRD lanes that admit new repair classes.

2.3.2 Creativity and Authority Live in Different Components

A second architectural commitment, which I have not seen articulated explicitly in the related-work literature, is that *creativity* and *authority* are deliberately assigned to different components:

Component	Nature	Authority
LLM worker	Creative, stochastic	Proposes
LLM judge	Advisory, stochastic	Comments
Promotion gate	Deterministic	Decides
Audit chain	Append-only	Remembers
Shepherd skill	Human verification automated	Escalates rare failures

The principle is short: *let the LLM be creative, but never let it be authoritative.* LLM workers generate candidate patches; LLM judges generate advisory commentary; the deterministic promotion gate decides whether those patches may affect production; the append-only audit chain is the load-bearing record of what was decided and why. Section 4.1 shows how this separation maps onto concrete components, and ğ8.6 reports the audit-chain hash-checkpointing that prevents silent gate-loosening.

2.3.3 Operational Scar Tissue as Curriculum

A third claim concerns what the loop is *trained on.* Most modern AI systems are trained on datasets, benchmarks, synthetic tasks, or self-play environments. Dataloop's RSI loop uses **production traces** as the teacher: warnings, failures, stale data, lifecycle inconsistencies, gate rejections, shepherd interventions, worktree failures, audit anomalies. The 31 registered harvester layers and 56-member `EventType` enum described in ğ5 are the input distribution; the failure clusterer is the sampler; the candidate-fix worker is the learner; the promotion gate is the evaluator. The compact claim is:

> *The curriculum for self-improvement is not human-labeled tasks. It is the system's own operational scar tissue.*

If this claim is correct, it has consequences for what scale of failure-mode coverage a self-improvement system needs before its loop becomes productive. Section 5.4 reports the inflection point at which trace-layer coverage in Dataloop crossed the threshold from "isolated incident handling" to "continuous cluster dispatch."

2.3.4 PRDs as Machine-Operable Product-Management Substrate

A fourth claim concerns the *substrate* of self-improvement. Most agent systems operate on *tasks.* Dataloop's Phase 2 feature pipeline operates on a **machine-readable product- management pipeline** — `proposed` → `approved` → `implementing` → `implemented` — with PRD repair lanes, metadata validation, editable-path declarations, acceptance criteria, decomposition into phases, lifecycle-debt detection, and promotion gates that read the PRD as authoritative. This is not "an agent doing code." It is closer to **autonomous software- organization design**: the agent treats product management, quality assurance, release engineering, and incident response as first-class subsystems with machine-operable artifacts. This paper does not claim that this is a complete software organization; it claims that the PRD-lifecycle primitive is a credible *direction* — and that the specific PRD-lifecycle primitive is, to our knowledge, undescribed in published agent-systems work.

2.3.5 Governed Complexity as Thesis

The five-system cognitive stack is intentionally complex. The temptation in agent-systems research is to claim that simplicity is a virtue and minimalism is a path. This paper rejects that path explicitly. Its thesis is the opposite, but not unfalsifiably so. This paper does *not* claim that "humans are infinitely complex, therefore AGI must be infinitely complex" — that argument is philosophical and unfalsifiable. This paper claims something narrower:

> *Human intelligence appears to be high-dimensional, stateful, socially embedded, memory-dependent, self-correcting, and multi-timescale. A plausible route to AGI may therefore require not only larger foundation models, but persistent cognitive systems that combine memory, perception, planning, reflection, tool use, error recovery, social feedback, and self-modification under stable governance.*

Complexity alone does not create intelligence. Complexity without governance usually produces fragility, not intelligence. The winning move, if there is one, is **governed complexity**: many subsystems, with sharp interfaces, audit trails, gates, invariants, and feedback loops. Dataloop is one attempt to explore that operating point empirically rather than philosophically.

2.3.6 Positioning Relative to the Monolithic-Model Story

Section 2.3 closes with a positioning statement. The dominant public story of AI progress over the last decade has been foundation-model scaling. A parallel research direction has emerged around agentic systems, self-improving scaffolds, automated agent design, evaluator-driven code evolution, and AI-driven scientific discovery (ADAS, DGM, Live-SWE-agent, AlphaEvolve, AI Scientist, AlphaProof). Dataloop belongs to that second direction, but pushes it toward *production operations* rather than benchmark-only improvement. This paper does not claim that this direction is correct and the scaling direction is wrong; it claims only that *the field is rediscovering that intelligence may live in the system around the model as much as in the model itself*, and that the present paper is one production case study of that hypothesis.

---

3. Evolutionary Path and Gap Analysis

The system documented in this paper is the result of a roughly 65-day arc starting from the founding PRD of 2026-03-17 and arriving at the operational state of 2026-05-21. This section maps that arc explicitly: which of the original requirements are now implemented (and how), what capabilities emerged that were never in the founding vision, and the commit-anchored timeline of how the project got from there to here.

3.1 Functional Requirements Coverage Matrix

The founding PRD specified eight functional requirements (F-001 through F-008) and six acceptance criteria. This paper maps each to its current implementation status:

Req	Name	Status	Evidence

F-001	Memory Taxonomy	**Implemented**	`Datakult_AGI_Utils/memory_locker/m` defines `MemoryKind` enum including `REWARD_SIGNAL` and other named kinds. `memory_kind` field is enforced in `Mem0_Master_Manager_Client.py` add/update paths.
F-002	Locker State Model (locked/mutable/quarantined)	**Implemented**	`is_locked_memory()`, `is_quarantined_memory()` helpers used at every read/write gate in `Mem0_Master_Manager_Client.py`.
F-003	Authoritative AITP Retrieval	**Partially implemented**	AITP uses a Mem0 panel + structured context pack. Original PRD called for a single canonical path; current implementation has two paths with documented coordination.
F-004	Event-Specific Writers	**Implemented**	`Datakult_AGI_Utils/memory_locker/w` exposes `build_reward_signal_write()` and analogous builders. AITP (`mem0_auto_save.py`) and Agent_v3 (`mem0_learning.py`) both consume them.
F-005	Mutation Decision Engine (ADD/UPDATE/MERGE/QUARANTINE/DELETE/NONE)	**Implemented**	`mutation_decision.action.value in {"quarantine", "update", "merge"}` decision gates appear at every mutating call site.
F-006	Reward Ledger	**Implemented**	`REWARD_SIGNAL` memory kind, `build_reward_signal_write()`, and the Agent_v3 user-feedback → reward_signal mapping documented in `Agents/Agent_v3/bug_fix.md`.
F-007	Memory Maintenance Loop	**Implemented**	`Mem0_Master_Manager_Client.consoli` is the maintenance primitive, invoked on scheduled cadence.

F-008	Locker Observability	**Implemented**	Tests at `tests/mem0/test_memory_locker.py` and `tests/mem0/test_memory_locker_cont` exercise mutation history and quarantine state. Operator inspection paths exist.

Acceptance Criterion	**Status**
AITP no longer relies on generic cycle summary text as its primary live Mem0 write source	**Met** — structured writers replaced the generic path
System can store and retrieve all 7 named memory kinds	**Met** — all 7 kinds + additional kinds (corruption_report, fold_summary, belief_update) implemented
At least one mutation path each for UPDATE, MERGE, QUARANTINE testable	**Met** — exercised by locker test suite
Retrieval provenance can explain why a memory was shown and what its locker state is	**Met** — provenance attached at write time, exposed via locker observability
Clear user approval and clear user rejection both produce reward updates	**Met** — `build_reward_signal_write()` is invoked from both signal paths
Operators can inspect quarantined memories and mutation history without querying Mem0 directly	**Met** — locker control-plane tests cover this path

Table 5: Founding-PRD requirements coverage. All eight functional requirements are at least partially implemented, with seven fully implemented. All six acceptance criteria are met. The founding vision is substantively complete.

3.1.1 The Memory Taxonomy as Implemented

The founding PRD specified an explicit memory taxonomy (F-001) but did not finalize the kinds. The current implementation enumerates ten kinds in `Datakult_AGI_Utils/memory_locker/models.py`:

Memory kind	**Purpose**	**Writer function**
`identity_profile`	Who Dataloop is talking to; user identity facts	(writer in AITP path)
`user_preference`	Persistent user-stated preferences	(writer in AITP path)
`task_outcome`	Outcomes of completed tasks (success/failure with evidence)	`build_task_outcome_write()`
`procedural_tool_use`	Reusable tool sequences that worked	`build_tool_execution_write()`

`reasoning_outcome`	Conclusions from System Two cycles	`build_reasoning_outcome_write()`
`reward_signal`	Positive/negative outcome reinforcement	`build_reward_signal_write()`
`self_reflection`	Insights about Dataloop's own behavior	`build_aitp_cycle_write()`
`belief_update`	Revisions to long-held beliefs	(writer in System Two path)
`corruption_report`	Memories flagged as inconsistent	(locker maintenance path)
`fold_summary`	Compressed historical context	(folding planner path)

Table 5b: Memory taxonomy as implemented. The first seven kinds map directly to founding-PRD acceptance criteria; the last three (`belief_update`, `corruption_report`, `fold_summary`) were added in subsequent design rounds without explicit founding-PRD mandate but consistent with its spirit. The shared base writer `build_memory_write_request()` constructs the underlying `MemoryWriteRequest` with the metadata + provenance + reward fields uniformly across kinds.

3.2 Capabilities That Emerged Beyond the Founding Vision

The current system contains capabilities that the founding PRD did not request, did not anticipate, and in some cases would have been impossible to specify in March 2026. This paper catalogs the major ones here because they constitute the bulk of subsequent engineering effort and the most novel contributions of the project:

Capability	**Founding PRD?**	**First commit**
Trace-harvester infrastructure (31 registered harvesters / 56 current `EventType` enum members)	No	Early April 2026, expanded continuously
Failure clustering with severity Œ recurrence ranking	No	Mid-April 2026
Cluster-and-dispatch worker pattern	No	2026-04-21 (Phase 2 launch)
Phase 2 Feature Implementation Pipeline	No	2026-04-21 (`feat(rsi): ship Phase 2 Feature Pipeline...`)
Phase 3 Karpathy-mode parameter optimization	No	2026-04-21 (`feat(rsi): expand Karpathy-Mode loop...`)
Promotion gate with locked-eval composite scoring	No	Iteratively, late April-May 2026
Append-only audit chain with hash checkpoints	No	Mid-April 2026
Worktree-based candidate isolation	No	Phase 2 launch
Self-healing supervisor with heartbeat monitoring	No	Late April 2026
Failure-recovery patch generation	No	May 2026

Autonomous proposal_watchdog	No	2026-05-07 (`feat(rsi): autonomous proposal_watchdog (operator-out-of-the-loop ASI loop)`)
Mixed-priority lane scheduler	No	2026-05-12
PRD-repair lane (auto-fix invalid PRDs)	No	Early May 2026
Codex App Server control plane	No	2026-05-16
Phase-decomposed feature dispatch	No	Mid-May 2026
Substantive-evidence promotion invariant	No	2026-05-07 (originated from operator feedback)
Promotion advisory judge	No	2026-05-07
Iteration feedback judge	No	2026-05-07 (advisory-only)
Cross-campaign recurrence admission	No	2026-05-12
Karpathy lane calibration / wave heartbeat	No	Late April 2026
Real module decomposition of `loop_runner.py` and `feature_scanner.py`	No	2026-05-04
Analysis trace mining (Layer 10)	No	2026-05-06
Bulk analysis scheduler automation	No	2026-05-10

Table 6: Capabilities that emerged beyond the founding vision. None of the items in this table appear in the original PRD's F-001 through F-008. Most are not even adjacent to the memory-locker framing. The project's surface area roughly *tripled* between mid-April and mid-May 2026, and that expansion was driven almost entirely by what RSI itself proposed and implemented.

The relationship between the founding vision and the current state is therefore not "the original plan, scaled up." It is closer to: *the founding PRD provided a viable substrate (structured memory, mutation discipline, reward signals), and a code-modifying capability emerged on top of that substrate that proved general enough to extend itself.*

3.3 Evolutionary Timeline

The following timeline anchors each major capability to its first appearance on `origin/main`. All commits are verifiable in the repository's git history under controlled reviewer access (see *Code and Data Availability*).

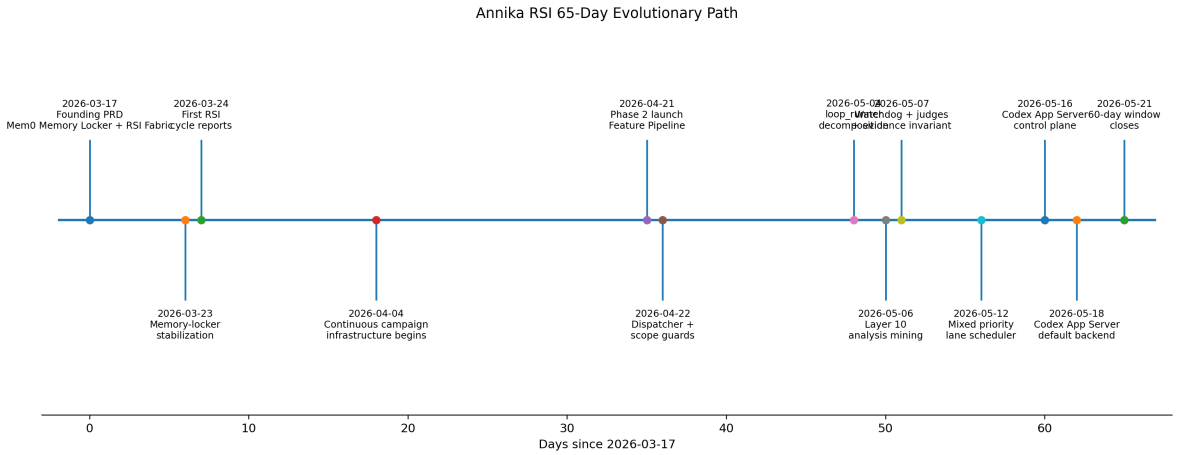

Figure 11: 65-day evolutionary timeline. Each entry is sourced from `git log` on `origin/main`. The chronology shows three distinct phases: (a) March 2026, foundation period focused on memory locker stabilization; (b) April 21–22 2026, inflection where Phase 2 (Feature Pipeline) and Phase 3 (Karpathy-Mode) launched together and the code-modifying loop became operational; (c) May 2026, expansion period where RSI began proposing and implementing its own structural improvements at scale.

The most concentrated burst of self-improvement infrastructure landed in a single 24-hour window on **2026-05-07**: the autonomous proposal_watchdog, the promotion advisory judge, the iteration feedback judge, and the substantive-evidence invariant. Before that day, the system needed operator intervention to reconcile dual-lifecycle PRD directories, review accepted gates, and steer retry tactics. After that day, all four of those needs had automated handlers.

The 2026-05-18 transition to the Codex App Server backend is the most recent structural change in the window. The previous codex-exec backend remains available as a rollback path. Claude remains in the outer backend/failover machinery, but current app-server configuration does not silently fall back inside the app-server lane; failures are meant to stay visible to RSI unless fallback is explicitly enabled.

---

4. The Cognitive Architecture of Datakult 2.0

Datakult 2.0 is a 24/7 always-on assistant whose architecture is deliberately structured as five distinct systems, each named after a layer of biological cognition. RSI is **System Four**, and its design follows from that placement.

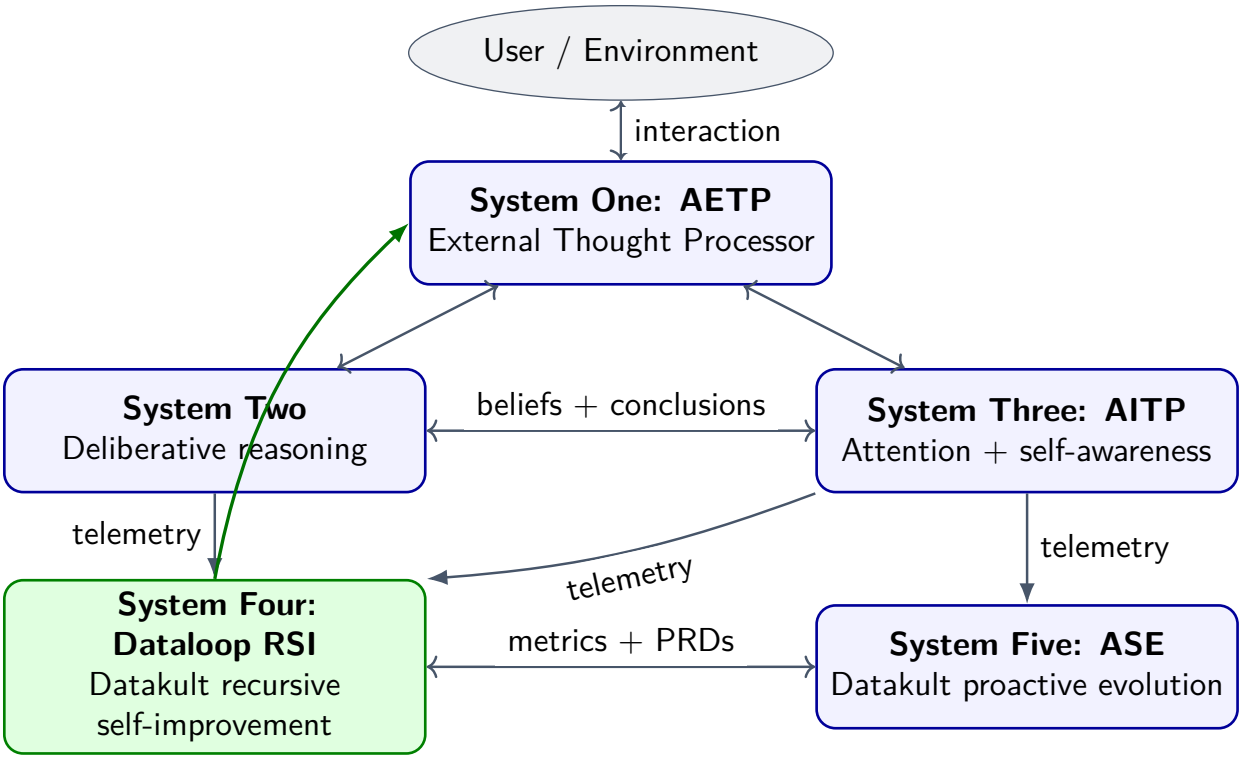

Figure 1: The five-system cognitive stack. AETP, System Two, and AITP form a closed loop for normal cognition. RSI and ASE observe the cognitive loop, detect deficiencies, and write patches back into it. RSI is reactive (responds to failures); ASE is proactive (proposes evolution). Both are designed to write to the same promotion gate.

Status caveat. During the 60-day reporting window, System Four (RSI) was empirically active and accounts for the commit-volume results reported here. **System Five (ASE) was architecturally specified but not empirically active**: the five-system stack is the design target. No ASE-originated PRD entered the `Future_Updates/proposed/` lifecycle in this window, and the ASE → RSI edge in the figure above should be read as a planned channel rather than an observed one. This paper discusses this explicitly in ğ8.9 (honest negative results) and treat "ASE goes live" as one of the milestones for the next reporting window in ğ11.

4.1 The Immune-System Framing

A common error in autonomous-improvement systems is to treat the improver as another optimizer, equivalent in priority to the systems being improved. This paper treats RSI as an *immune system*: it sits at a different abstraction level, observes the operational substrate, and has special authority to write to it but only through audited, gated channels. This framing has three engineering consequences:

1. **Trace-first.** RSI cannot read live state directly; it reads harvested execution traces. This forces the operational systems to *emit* their state in structured form, which has second-order benefits for debuggability.
2. **Bounded authority.** RSI's edits are restricted to an editable-paths allowlist defined per-feature. It cannot modify the promotion gate without going through the gate.

3. **Append-only audit.** Every promotion decision is recorded in an append-only JSONL audit chain. Forensics, post-mortems, and regression-tracking all read from this chain.

4.2 Why RSI is not System Five

Many in this literature collapse the "self-improving" agent into one component. This architecture separates **reactive** improvement (RSI / System Four) from **proactive** evolution (ASE / System Five) because they have different failure modes and different gates. RSI must not invent novel features — that is a separate authority. ASE must not commit to main without RSI's locked-eval gate. The separation is *organizational*, but in code it manifests as two different lifecycle directories (`Future_Updates/proposed/` vs `Future_Updates/approved/`) and two different sets of pre-promotion checks.

---

5. The RSI Loop

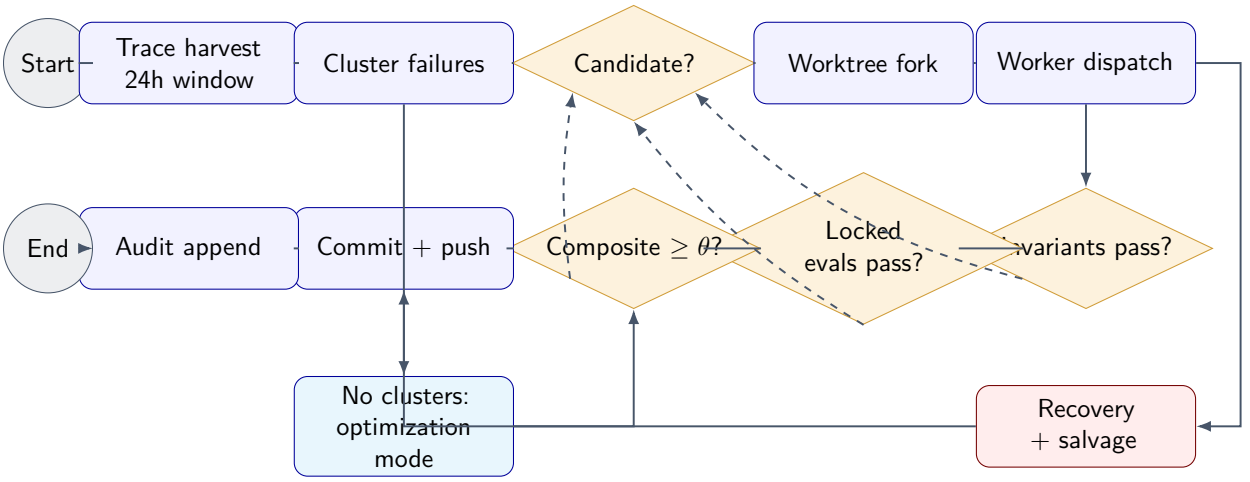

Figure 2: The RSI loop as a finite-state machine. The loop has three entry conditions (trace-driven, Karpathy-driven, and proactive from ASE), three exit conditions (commit-and-push, recovery-and-push, or abort-and-retry), and one durable side effect (the audit append).

5.1 Phase 1 — Bug-fix loop

The bug-fix loop is the foundational case. A trace harvester runs over the last 24 hours of execution traces from 28 instrumentation layers (system one, system two, AITP, MCP servers, panels, voice pipeline, etc.) and clusters failures by similarity. Each cluster receives a *failure hypothesis*: a short text description of the apparent root cause. The cluster with highest severity Œ recurrence is selected as the candidate target. A worktree is forked from the current `origin/main`, an LLM-based worker is dispatched into the worktree with the hypothesis and a strict editable-paths allowlist, and the worker produces a candidate patch.

5.2 Phase 2 — Feature pipeline

Phase 2 is more interesting because it shows RSI doing *deliberate* engineering, not just bug-fixing. Features are represented as PRDs (Product Requirements Documents) in `Future_Updates/`. Each PRD has a lifecycle:

`proposed/` `approved/` `implementing/` `implemented/`

A *proposer* (auto-generated from trace clusters that exceed a recurrence threshold, or from human operators, or from ASE) writes a PRD to `proposed/`. A *watchdog* validates the PRD metadata and moves valid PRDs to `approved/`. The *dispatcher* forks a worktree, hands the approved PRD to a worker, and the worker implements it. On success the PRD graduates to `implemented/`; on failure it returns to `approved/` with a retry count.

5.2.1 The Proposal Watchdog

The proposal watchdog is the gatekeeper between `proposed/` and `approved/`. It runs on a heartbeat and performs the following checks on each PRD bundle (typically `PRD.md`, `SCOPE.md`, `IMPLEMENTATION_PRD.md`, `IMPLEMENTATION.md`, `analysis.md`):

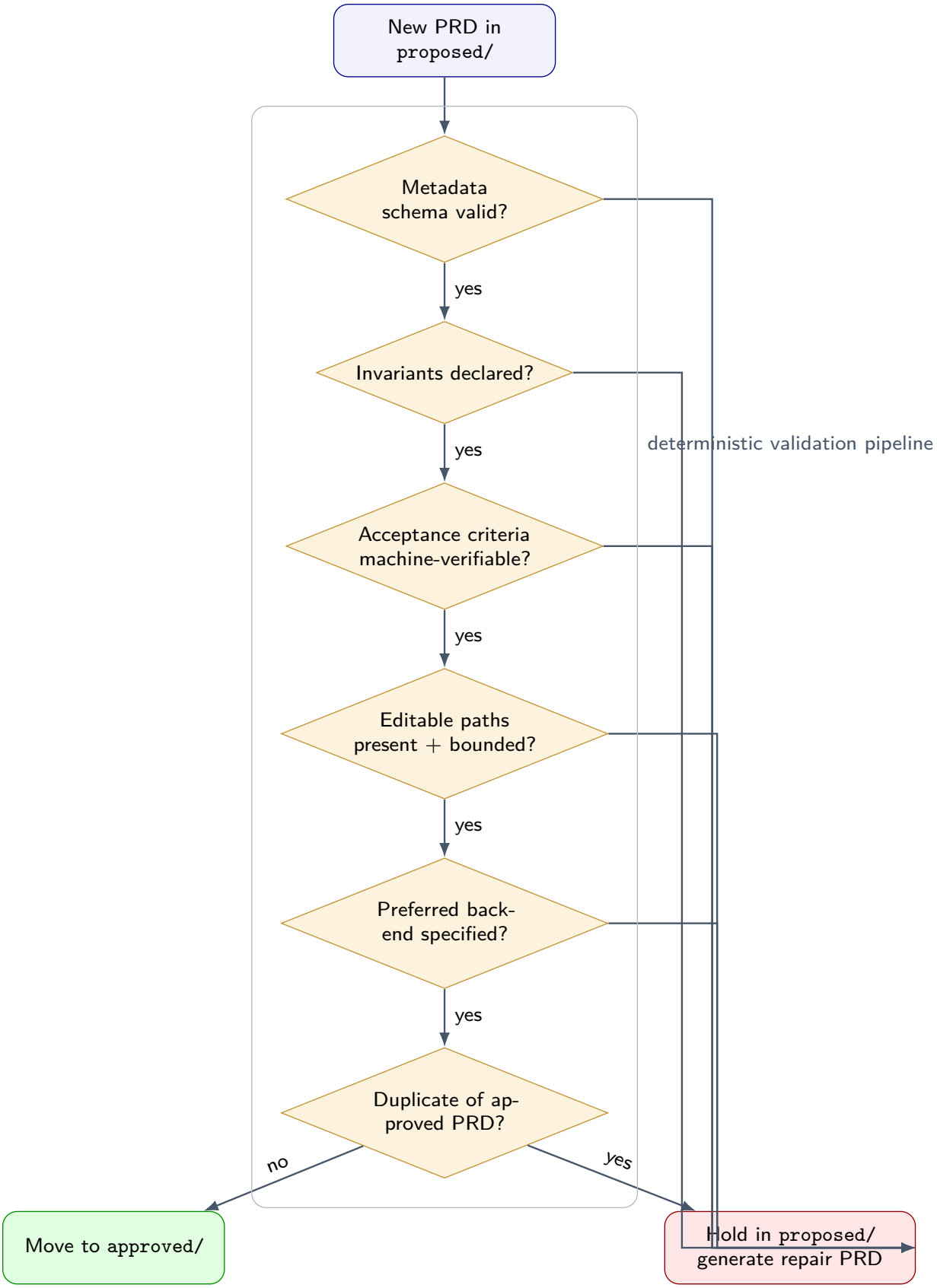
New PRD in proposed/
Metadata schema valid?
yes
Invariants declared?
yes
Acceptance criteria machine-verifiable?
yes
Editable paths present + bounded?
yes
Preferred back-end specified?
yes
Duplicate of approved PRD?
deterministic validation pipeline
no
yes
Move to approved/
Hold in proposed/ generate repair PRD

Figure 4.5: The proposal watchdog. A PRD must satisfy six metadata checks before promotion to the dispatch queue. PRDs that fail any check remain in `proposed/`, and an *auto-repair PRD* is generated whose `editable_paths` are constrained to the failed PRD's own metadata files. This means the system can *repair its own proposals* without operator intervention. The heartbeat ensures the watchdog continues even when no operator-authored PRDs arrive; trace-harvester-generated PRDs keep the queue alive.

5.2.2 Worker Tactics and Progressive Attempt Strategy

When a worker is dispatched against a PRD, it is assigned a *tactic* from a fixed library (`Master_Prompts/rsi/featu`
The tactic library currently contains six entries; the dispatcher selects one per attempt according to a progressive-narrowing policy.

Tactic ID	Description	Switch criteria
`full_scope_baseline`	Implement the full approved PRD as written	First attempt or clean prior history
`subsystem_first`	Implement one subsystem end-to-end with explicit deferrals for the rest	Prior full-scope attempt failed or produced weak evidence
`test_first`	Write gate-relevant tests first, then implement smallest code	Prior attempt lacked acceptance-criteria or behavioral proof
`prompt_only`	Modify only prompts/configs/tests needed for runtime behavior	Affected paths dominated by `Master_Prompts/` or config
`interface_first`	Define stable interfaces/types/contracts before filling in	Failures show cross-module shape mismatch
`narrow_to_smallest`	Pick smallest high-confidence path, document the rest as follow-up	Late retry after broader tactics failed

Table 8: Phase 2 Worker Tactics. Each PRD gets up to five attempts. The dispatcher logs `tactic_id` in the audit chain for every attempt, enabling post-hoc analysis of which tactic succeeded for which PRD class. The strategy is *deterministic*: given identical prior attempt history and identical PRD metadata, the dispatcher selects the same next tactic.

The 5-attempt progressive strategy (`feature_attempt_strategy.py`) sequences as: full scope → boosted full scope → priority subsystem → boosted most-tested subsystem → one high-confidence file. Tier demotion to `proposed/_returned/` requires five consecutive rejections in this sequence; this prevents premature abandonment of borderline PRDs while bounding total worker investment.

5.3 Phase 3 — Karpathy-mode

When there are no active bug clusters, RSI shifts to Karpathy-mode: a parameter-tuning loop modeled on the iterative training-recipe philosophy of Karpathy (2019). Calibration lanes sweep

prompt parameters, model parameters, and gate thresholds against a fixed evaluation harness, with the same locked-eval promotion gate applied to optimization candidates.

Each Karpathy lane declares the **Karpathy Triplet**:

lane_id**:** system_one_response_quality *# canonical lane name* editable_glob**:** "Master_Prompts/..." *# files the meta-agent may modify* primary_metric**:** *# scalar that drives keep/revert* name**:** judge_score source**:** sys_two_module.conclusion_generator.grade_response provider**:** judge_holdout *# judge_holdout | runtime_telemetry | static_analysis* direction**:** maximize holdout_size**:** 30 observational_metrics**:** [...] *# recorded but non-gating* invariants**:** [...] *# fail any → revert* time_budget_seconds**:** 300 *# dispatch timeout* wave_branch_strategy**:** campaign_branch *# direct_to_main | campaign_branch* canary_strategy**:** manifest_holdout_required

The lane catalog (`config/rsi/lane_metrics.yaml`) ships with **24 explicit-provider lanes** covering: System One response quality, System One response strategy, AETP notification routing, AITP memory extraction, Enhanced Prompt context quality, judge holdout, runtime telemetry, prompt-only edits, panel canary fixtures, task-manager dedupe, AITP attention routing, observability gates, safety gates, MCP tool dispatch, UI panels, and several others. The provider field gates the lane: `judge_holdout` lanes require canary holdout fixtures before they can optimize; `runtime_telemetry` lanes fail closed when metrics are absent; `static_analysis` is reserved for structural metrics rather than silent fallback.

5.3.1 Phase 4 — Lane Tuning Proposal Pipeline

Phase 4 (added 2026-05-06) is a *meta-loop* on top of Phase 3: it generates structural proposals for *adjusting* the Karpathy lane definitions themselves. When a Phase 3 lane exhibits `bringup_incomplete` status across multiple waves (its baseline does not reliably pass before optimization attempts can run), Phase 4 proposes threshold recalibration PRDs backed by statistical evidence.

For example, the AITP memory extraction lane was recalibrated with this evidence pack (per `@module-rsi-phase4.mdc`):

```
lane: aitp_memory_extraction
baseline_mean: 0.0164
baseline_sigma: 0.0163
required_pass_rate: 0.5000
recommended_threshold: 0.5000
sample_count: 5
applied bringup_pass_rate_threshold: 0.50
```

Writes to `config/rsi/lane_metrics.yaml` are *protected*: a worker may only modify the file when the dispatch is operator- approved *or* the audit context records `kind: lane_tuning_proposal`. This prevents Karpathy itself from rewriting its own gates outside the proposal pipeline. The `is_protected_lane_config_write` guard enforces this; a regression test pins it.

5.4 The Promotion Gate

The promotion gate is the single most important component of the system, because it is what prevents the loop from corrupting itself. In the Phase 2 feature pipeline, the completion-promise layer evaluates candidates against invariant records such as:

Invariant	Type	Description
`pytest_passes`	blocking	All affected tests pass; exit code 0
`scoped_tests_pass`	blocking	Only tests touching the candidate's paths are required
`scoped_test_targets_present`	blocking	Worker must have authored at least one scoped test
`lint_passes`	blocking	Ruff E501 etc. clean on changed files
`centralized_managers_preserved`	blocking	No direct `import redis`; uses Redis_Master_Manager_Client
`no_inline_prompt_sprawl`	blocking	No hardcoded LLM prompt strings outside Master_Prompts
`acceptance_criteria_covered`	warning	All PRD acceptance criteria have evidence
`file_inventory_complete`	warning	PRD file_inventory matches actual diff
`generated_skill_projection_preserved`	warning	Skill projections not silently reverted

The lower-level `scorer.py` hard-gate primitive is smaller and more mechanical: it rejects protected-file edits, locked-eval regressions, newly introduced failures, and removed tests. The feature-pipeline invariants above are recorded as richer gate evidence around that scoring primitive rather than being defined verbatim inside `scorer.py`.

A *composite score* is computed from replay wins, the locked-eval pass rate, and a *bugfix-precision* sub-score.

Formal definitions. Let C_i be the failure cluster targeted by iteration i, P_i the candidate patch produced, and F_{pre}, F_{post} the sets of failing tests before and after applying P_i. Define:

- **Cluster-fix indicator** $1[C_i\text{resolved}]$: 1 if all failures attributed to C_i appear in $F_{pre}F_{post}$; 0 otherwise.
- **Regression indicator** $1[\text{regression}_i]$: 1 if $F_{post}F_{pre}$; 0 otherwise.
- **Bugfix-precision** over a window W of iterations:

$$\text{BP}(W) = \frac{\sum_{iW} 1[C_i\text{resolved}](1 - 1[\text{regression}_i])}{|W|}$$

That is: BP is the fraction of iterations that fix their targeted cluster *and* introduce no new failures.

Composite score is a weighted sum of seven component sub-scores. The current implementation in `Datakult_AGI_Utils/rsi/scorer.py` defines the composite as:

$$\text{Composite}_i = 0.35R + 0.20E + 0.15B + 0.10L + 0.10T + 0.05D + 0.05K$$

where:

Term	Symbol	Definition
Replay-win rate	R	`replay_wins / replay_total` — fraction of replay scenarios in which the candidate succeeded
Locked-eval pass	E	`locked_eval_passed / locked_eval_total` — fraction of locked tests passing on the candidate
Bugfix precision	B	`bugs_fixed / bugs_targeted` — fraction of targeted clusters actually fixed (alias of BP above)
Latency score	L	$max(0, 1 - \text{latency/budget})$
Token-cost score	T	$max(0, 1 - \text{tokens/budget})$
Diff-minimality	D	$max(0, 1 - (\text{added} + \text{removed})/\text{MAX\_DIFF\_LINES})$
Rollback confidence	K	Penalty for excessive `files_changed`, capped at 1.0

The seven weights sum to 1.00, so $\text{Composite}_i[0, 1]$. Promotion requires (i) all hard gates pass *and* (ii) $\text{Composite}_i \geq 0.6$; the band $[0.3, 0.6)$ is flagged for review; below 0.3 is auto-rejected. Hard gates are checked independently of the composite and a single hard-gate failure overrides any composite score (the candidate is rejected even if the composite would otherwise clear 0.6).

Version-pin note. During the early portion of the reporting window the deterministic gate used a simpler 3-component composite over invariants, locked-eval pass rate, and bugfix precision with weights $(0.4, 0.4, 0.2)$ and threshold 0.65; the 7-term composite documented above was in effect for the bulk of the window and is what a current clone of `Datakult_AGI_Utils/rsi/scorer.py` will show. This paper treats the 7-term formula as canonical for this paper and flag any earlier-window scorecard explicitly when this paper cites one. Appendix B is updated to the current configuration; Appendix D ğD.3 gives the exact command to print the in-effect threshold and weights on any clone.

5.4.1 LLM-as-Judge Quality Scoring

In addition to the deterministic invariants and locked-eval pass rate, the gate consults an LLM-as-judge quality scorer on a selected subset of candidates. The judge grades the candidate against five quality dimensions:

Dimension	What it measures
Accuracy	Did the patch address the actual bug/feature, not a symptom?
Anticipation	Does the patch handle edge cases the worker should have seen?

Hygiene	Does the patch leave the codebase cleaner than it found it?
Conciseness	Is the patch minimal for what it accomplishes?
Provenance	Are claims about prior work or tests verifiable from the diff?

Table 13: LLM-as-judge quality dimensions. Each dimension scores 0-5; total is normalized to 0-25. The judge's score is an *observational* sub-score: it informs the composite-score trend and is recorded in the audit chain, but it does not gate promotion on its own. I chose this deliberately: LLM-as-judge results are non-deterministic, and the gate should not depend on non-determinism in the gate's accept/reject decision. The judge provides signal for the operator and for Karpathy-mode tuning, not for production safety.

Implementation note. The `sys_two_module.conclusion_generator.grade_response` path was originally designed in March 2026 as the judge entry point. That implementation was deferred during early development; the current judge runs via `Datakult_AGI_Utils/llm_adapter` with prompt alias `rsi.quality_judge`. Memory in `Datakult_AGI_Utils/rsi/bug_fix_12.0.md` documents the deferral and current rewiring.

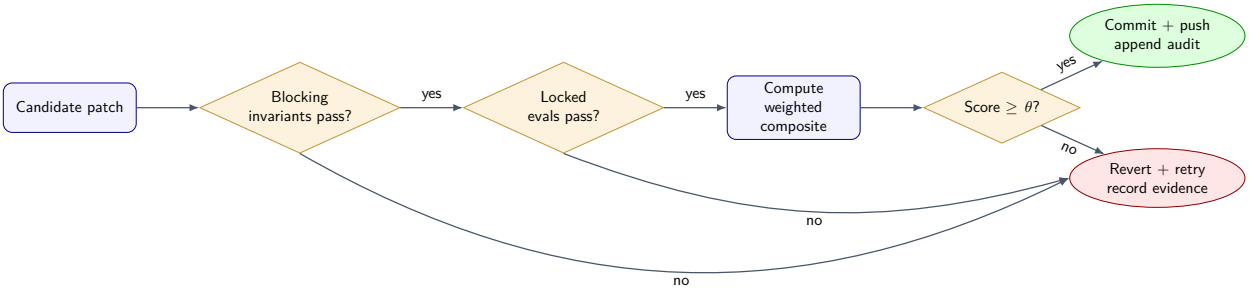

Figure 3: The promotion gate. Three sequential filters, each more expensive than the last. The cheap blocking invariants run first to fast-fail bad candidates; locked evals (full pytest suite) run only on candidates that pass invariants; composite scoring runs only on candidates that pass both. The gate is deterministic and reproducible from the audit chain.

5.5 Self-Healing

The supervisor process (`rsi_supervise.sh`) and the watchdog heartbeat monitor implement a self-healing layer one level above the RSI loop. If the loop wedges (no progress for N seconds), if a worker times out, if the backend hits a rate limit, or if a campaign's worktree falls out of sync, the supervisor restarts the appropriate component without operator intervention. This recovery layer is itself in-scope for RSI improvement: when a new failure mode is observed, an RSI iteration adds a detector and recovery path for that mode.

5.6 Supporting Mechanisms

The loop described in ğ5.1-ğ5.5 depends on four supporting mechanisms that each warrant explicit description because they were each load-bearing additions during the 60-day window and because reproductions in other systems would need analogues of each.

5.6.1 Trace Harvester Enumeration

The trace harvester layer is materialized as the `_HARVESTER_REGISTRY` tuple in `Datakult_AGI_Utils/rsi/trace_e` In the current codebase the registry contains **31 harvester layers**. The current `EventType` enum at `Datakult_AGI_Utils/rsi/trace_schema.py` contains **56 members**, including historical and reserved event classes that may not be emitted in every 24-hour harvest window. The table below is therefore a representative subset of event classes and source subsystems, not a claim that exactly 31 `EventType` values were emitted during the reporting window. Readers wanting the exact current registry and enum counts should run the commands in Appendix D ğD.2.

Class	Source subsystem	Captures
`ERROR_LOGGED`	All modules	Exceptions, error-level log lines
`WARNING_LOGGED`	All modules	Warning-level log lines
`BUG_FIX_LOGGED`	Per-module `bug_fix.md`	Recorded fix narratives
`AGENT_LOG`	`Agents/`	Agent_v3 lifecycle and decisions
`WORKER_LOG`	RSI workers	Per-iteration worker activity
`JOBWORKER_LOG`	MCP_Server job queue	Tool dispatch and job execution
`SYSTEM_ONE_LOG`	AETP	Fast-path response generation
`SYS_TWO_COMPONENT_LOG`	System Two	Belief/conclusion/emotion components
`TASK_MANAGER_LOG`	Task manager	Task lifecycle and assignment
`OPENAI_AGENT_LOG`	OpenAI-driven agents	Provider-level decisions
`LLM_RESPONSE`	LLM adapter	All LLM call outcomes with provenance
`REASONING_TRACE`	System Two	Structured reasoning trajectory digests
`BELIEF_FORMED`	System Two	Belief updates
`EMOTION_RECORDED`	System Two	Emotional state observations
`QUESTION_GENERATED`	System Two	Generated reasoning questions
`TOOL_COMPLETED`	MCP tools	Tool-execution outcomes
`ANALYSIS_FINDING`	`analysis.md` mining (Layer 10)	Module-level intention-vs-reality findings
`ANALYSIS_SNAPSHOT`	Bulk analysis scheduler	Periodic module-health snapshots
`PANEL_CANARY`	Enhanced Prompt panels	Final panel-budget trim observations
`QUALITY_SCORE`	Quality harness	Composite/sub-score readings
`RSI_PERFORMANCE`	RSI itself	Self-improvement coverage metrics
`METRIC_RECORDED`	Prometheus	Time-series metric snapshots
`LATENCY`	All instrumented paths	Per-path p50/p95/p99 latencies
`COMPONENT_HEALTH`	Health checks	Service-level liveness/readiness
`LAYER_HEARTBEAT`	Per-layer heartbeats	Confirms a layer is alive but quiet
`STALE_DATA`	Freshness checks	Data older than its freshness budget

STUCK_PROCESS	Process watchdog	Long-running processes without progress
FLOW_ANOMALY	Cross-module flow	Unexpected transitions in component graphs
CONSISTENCY	Cross-store consistency	Redis <-> disk <-> memory disagreements
UI_CONSOLE_ERROR	Browser console	UI client-side errors
UI_HTTP_ERROR	UI API surface	4xx/5xx responses
UI_WEBSOCKET_ERROR	UI WebSocket	Connection/protocol errors

Table 9: Representative `EventType` values emitted by RSI harvester layers. This is a curated subset of the current 56-member `EventType` enum at `Datakult_AGI_Utils/rsi/trace_schema.py`; the precise registry and enum sizes are printable via the commands in Appendix D ğD.2. Each `EventType` has its own normalizer, structured payload schema, and ranking weight in the cluster scorer. The `ANALYSIS_FINDING` row was added 2026-05-06 and substantially improved cluster quality by surfacing intention-vs-reality mismatches that error logs alone cannot detect.

The harvester runs on a 24-hour rolling window by default (`harvest_traces(hours_back=24)`). Window size is a Karpathy lane parameter — wider windows surface chronic patterns at the cost of recency sensitivity; narrower windows respond fastest to acute issues.

5.6.2 PRD Repair Lane Mechanics

When a Phase 2 PRD fails preflight validation, it does not disappear — it enters the *PRD repair lane*. The repair lane addresses three classes of mechanical metadata defects:

Defect class	**Symptom**	**Repair**
`missing_yaml`	No `### RSI Feature-Pipeline Metadata` block	Inject default metadata block synthesized from PRD body
`empty_affected_paths`	Metadata block present but `affected_paths:` list empty	Mine paths from PRD body text via regex on code-reference patterns
`missing_invariants`	No feature-specific `invariants:` declared	Extract baseline invariants from PRD acceptance-criteria language

Table 10: Repairable PRD defects. Note that *missing acceptance criteria* is **not** a repairable defect — the repair lane refuses to fabricate AC markers, and PRDs with fewer than three AC markers stay rejected. This is a deliberate guardrail against AC drift: acceptance criteria are the contract; the system will not write its own contract.

The repair lane is atomic: it writes a candidate body to a filelocked temporary path, validates the candidate against `validate_prd_prescan()`, and *only on validator pass* atomically replaces

the original PRD. Failed candidates are discarded without modifying the source. Repair attempts append structured JSON to `data/rsi/prd_repair_attempts/<feature>/log.json` with the full `RepairResult` Pydantic model (defects_found, defects_fixed, candidate vs persisted body hashes, validator reasons, rollback reasons).

After two consecutive repair failures, the lane escalates by emitting *narrower subsystem PRDs* via `prd_narrower_scope_emitter.py`: the parent PRD is marked `blocked_pending_subsystem_splits`, child PRDs are created in `Future_Updates/proposed/` with restricted scope, and the parent auto-promotes only after all splits land in `implemented/`.

5.6.3 Subsystem Decomposition for Oversized PRDs

PRDs whose `affected_paths` exceed the `DECOMPOSITION_PATH_LIMIT = 40` threshold trigger automatic *subsystem decomposition*. The decomposer groups paths by their top-level subsystem prefix (`Agent_Tools/`, `meta_state/`, `User_Interface/`, etc.) and emits a deterministic feature ID per phase using the convention:

```
<parent_feature>__phase_<subsystem_slug>
```

For example, a 75-path PRD touching `Agent_Tools/`, `meta_state/`, and `User_Interface/` becomes three child features: - `parent_feature__phase_agent_tools` (paths in `Agent_Tools/`) - `parent_feature__phase_meta_state` (paths in `meta_state/`) - `parent_feature__phase_user_interface` (paths in `User_Interface/`)

Each phase is dispatched independently with its own `editable_paths` slice. The parent's invariants pass through unchanged. The dispatcher records `current_phase` and `max_phases` in `iteration-NNN-context.json` so audit replay can reconstruct which slice was active at any iteration.

Early phases (e.g., cursor or github work that doesn't touch runtime code) commonly trigger `scoped_test_targets_present` warnings; the `disposition_hint=promote_next_iteration` field instructs the gate to trust these and continue. Path normalization strips leading `.` so a PRD specifying `.cursor/` matches the gate's expected `cursor/`.

5.6.4 Cross-Campaign Recurrence Admission

When the same failure pattern appears in multiple consecutive campaigns, the meta-synthesis layer can emit a `cross_campaign:*` wrapper that bundles the recurrent children into a single synthetic cluster. To prevent fixture/test/demo noise from generating spurious wrappers, all candidate wrappers pass through `cross_campaign_pattern_admission.py`, which classifies evidence into four classes:

Evidence class	Meaning	Admission decision
`GROUNDED`	Child has real files, commits, source PRD affected paths, OR runtime provenance	**ADMIT**

`SPARSE_FIXTURE`	Child id contains demo/test/fixture/synthetic markers AND no grounded evidence	**HOLD** (admission deferred until grounding appears)
`SPARSE_UNKNOWN`	No grounded evidence and no fixture markers — ambiguous	**HOLD**
`MALFORMED`	Evidence cannot be normalized (missing required fields, parse errors)	**REJECT**

Table 11: Cross-campaign admission decisions. The admission gate is a *deterministic, side-effect-free* classifier. It does not consume the candidate wrapper — it only emits a decision plus structured reason codes that downstream consumers may use to defer, drop, or admit. Each HOLD or REJECT decision records operation IDs, counts, and fingerprints so the operator can audit (or RSI can later re-evaluate) the rationale.

Memory entry `2026-05-12 cross-campaign recurrence admission` documents the introducing incident: a `demo`-named child id had grounded evidence (real files, real commits) and was being held by an earlier version of the gate that rejected on the name alone. The current implementation evaluates *grounded-evidence-first*: if grounded evidence is present, the candidate is admitted even when the id contains fixture markers.

5.6.5 Codex App-Server Thread Semantics

As of 2026-05-18, the default worker backend is the Codex App Server, accessed through `codex_app_server_operat`
The operator treats the backend as an LLM executor inside a harness rather than a fire-and-forget worker process. The state model:

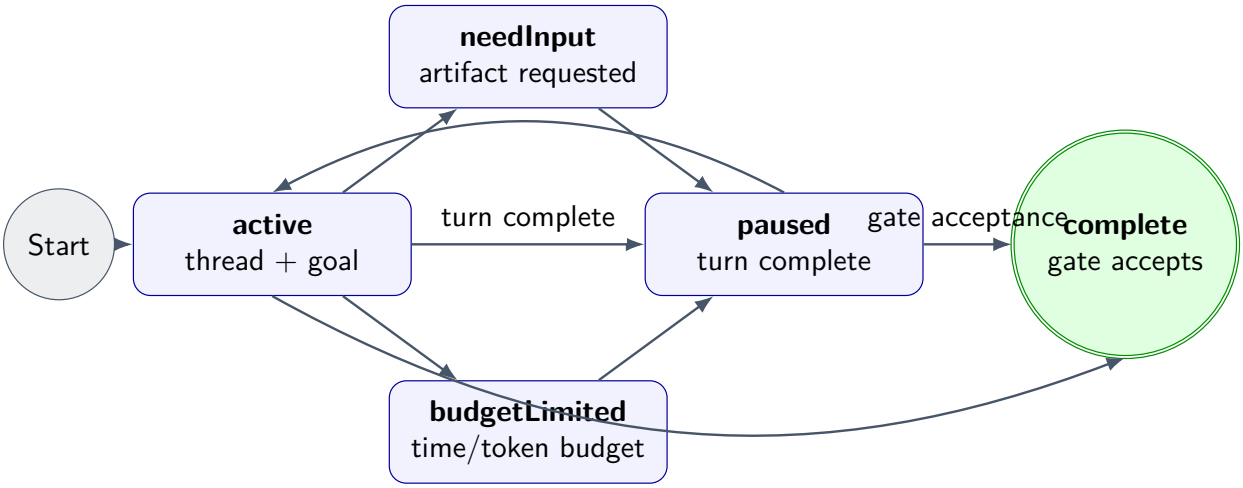

Figure 12: Codex App Server thread lifecycle. Thread `goal_status` transitions are gated by the *outer* RSI promotion gate, not by the worker itself. A worker can never self-mark "complete";

only gate acceptance progresses the goal. Failed gates send `GateFeedbackPacket` items back as new input on the same thread, enabling iterative refinement without losing worker session state.

The `CodexThreadRegistry` (defined in `codex_thread_registry.py`) tracks one record per `(feature_id, objective_hash)` pair. Thread reuse occurs when the same objective is dispatched a second time: the registry surfaces the prior thread and the operator resumes from `paused` state rather than spawning a new thread. This is what produces the 15-20% per-iteration speedup observed at the 2026-05-18 backend transition (per Table 7 row).

Need-input events are resolved via `resolve_need_input_from_artifacts()`: the operator inspects the worker's `NeedInputPacket`, locates the requested artifact in the worktree (or generates it from existing RSI state), and returns the resolved content as a new input item. The worker sees this as a natural continuation, not as an external interruption. If no resolution exists, the operator marks the thread `budgetLimited` and the supervisor's escalation policy takes over.

5.6.6 Iteration Feedback Judge (Advisory-Only)

When a Phase 2 feature attempt is rejected by the promotion gate, the rejection enters `iteration_feedback_judge` an *advisory-only* LLM reviewer configured via `Master_Prompts` and the centralized LLM adapter (call alias `rsi.iteration_feedback_judge`). The judge:

1. Reads the rejected attempt's gate verdict, invariant evidence, and worker diff
2. Produces a structured `IterationFeedbackResult` containing:
   - A *suggested_tactic* drawn from the 6-tactic library
   - A *primary_failure_class* drawn from a fixed taxonomy
   - A *human-readable suggestion* capped at 120K characters
3. Persists the result as a capped (last-N) history list in the feature implementation state (`judge_feedback_history`)
4. Writes an `iteration_judge_feedback` audit record
5. Returns control to the dispatcher

The dispatcher's next attempt injects the most-recent feedback into the worker prompt via `cycle_prompt_builder`. The judge's `suggested_tactic` *steers* the next tactic selection unless the immediately previous attempt also failed with the same tactic (in which case the dispatcher falls through to the deterministic progressive sequence).

The judge never promotes or blocks. Its authority is strictly advisory. This separation matters: a single LLM-as-judge that could also gate promotion would centralize too much power in a non-deterministic component. By keeping the judge advisory and the gate deterministic, the system gains LLM-mediated context-sensitivity without losing the audit-traceable safety property.

The judge has a default budget of 30 seconds and a token ceiling of 120,000 input characters. Beyond budget, the call returns a `JUDGE_UNAVAILABLE` result and the dispatcher proceeds without advisory feedback. This is by design — the loop must continue when the judge is unhealthy, since the judge is not load-bearing for correctness.

---

6. Methodology

6.1 Data sources

All measurements in this paper come from the canonical git history of the Datakult 2.0 repository at `origin/main`, augmented by structured run-artifacts written by the RSI supervisor:

Source	Path	Description
Git history	`.git/`	Authoritative commit log on `origin/main`
Cycle reports	`data/rsi/reports/*.md`	Per-iteration reports
Run heartbeat	`data/rsi/runs/<id>/monitor/heartbeat.json`	Live campaign state
Audit chain	`data/rsi/audit/promotion_audit.jsonl`	Append-only promotion log
Recovery patches	`data/rsi/runs/<id>/recovery/*.patch`	Cherry-pick safety nets
Cursor rules	`.cursor/rules/*.mdc`	Static architectural rules
Code review docs	`CLAUDE.md`, `AGENTS.md`	Operator-authored handbook

6.2 Measurement window

The reporting window is **2026-03-23 to 2026-05-21 inclusive**, which is 60 calendar days end-to-end. March 23 was chosen as the first day on which non-trivial RSI-attributable activity was recorded. The window was chosen because it covers (a) the period during which the supervised continuous-mode runner was active, (b) the introduction of the shepherd-as-operator pattern, and (c) the introduction of the Phase 4 proposal pipeline.

6.3 Classification

Commits in the window were classified by their commit-message prefix:

Class	Pattern	Interpretation
structural	`fix(rsi)`, `feat(rsi)`, `refactor(rsi)`, `fix(shepherd)`, and matching shepherd feature/refactor prefixes	RSI improving its own machinery
auto	contains `[auto]`	Worker-authored, gate-approved, but absent from the current first-parent subject history
infra	`fix(tmux)`, `chore(skills)`, `chore(runtime)`, `fix(ui)`	Adjacent infrastructure
other	analysis-refresh, workflow-master projection, manual edits	Non-RSI work

This classification is intentionally conservative: structural RSI/shepherd prefixes are counted because they are reproducible from `git log -first-parent` subject lines. The broader `[auto]` worker-output bucket used in earlier internal analyses is not reported as a headline number here because the current committed classifier and first-parent subject history do not expose it cleanly. Other categories may contain RSI work but are not counted as such.

6.3.1 The Health Scorecard (8 Dimensions)

In addition to per-cycle composite scores, RSI maintains a *System Health Scorecard* across 8 dimensions, sampled each iteration. Each dimension scores 0-20; the total normalizes to 0-100 and is stored as `RsiState.health_score`. The weakest dimension becomes the target for the next iteration, with rotation to the next weakest if the same dimension stays weakest for two consecutive rounds.

Dimension	Weight	What it measures	Source
Error Freedom	20%	P0/P1/P2 error counts across all logs	Log scan + Prometheus `rate(errors_total)`
Anticipation Quality	15%	Does AITP detect issues proactively?	AITP cycle logs, task creation patterns
Response Quality	15%	AETP accuracy + tone + concision	Prompt/response logs, System Two conclusions
Test Coverage	10%	Locked eval count + pytest coverage	`pytest -collect-only -m locked_eval`, coverage
Observability	10%	% modules with Prometheus + structured logging	Registry scan, log file inventory
Documentation Freshness	10%	% `bug_fix.md` files updated within 30d	File timestamps, content scan
Genome Hygiene	10%	Stale configs, orphan prompts, unused aliases	Config scan, import analysis
Runtime Intelligence	10%	Response speed + reasoning depth + tool efficiency	Prometheus metrics

Table 14: Health Scorecard dimensions. Score bands per dimension: Poor 0-5, Good 11-15, Excellent 16-20. Campaign #8 (2026-04-23) reported a final health score of 47 from a starting score of 31 across iterations 3-7 — the kind of trajectory the scorecard is designed to surface. The weakest-dimension-targeting policy implements a discrete form of policy iteration: improve where you are worst, re-measure, repeat.

6.4 Velocity computation

For each two-week bucket, this paper counts structural commits and divides by bucket length in days. The bucket boundaries align with the campaign's natural milestones (W1-W2 was startup,

W3-W4 was an extended quiet period, W5-W6 was the inflection when supervised continuous mode came online, W7-W8 was sustained operation, and W9-W10 is the current accelerating window).

6.5 Source-code measurement

Total source LOC is measured via `git ls-files` enumeration and per-file `wc -l`. Per-module Python LOC is measured by restricting `git ls-files` to a module prefix. The aggregate 60-day delta is measured by aggregating `git log -shortstat` insertions and deletions.

6.6 Threats to validity

Section 10 discusses limitations. In summary: (1) machine-generated JSON artifacts inflate LOC counts; (2) the trace harvester's clustering threshold has been tuned during the window, which may affect cluster-count comparisons; (3) the operator (one of the authors) was active during the window, and operator commits are filtered out only at message-prefix level.

6.7 Operator Intervention Half-Life (OIH) — A Candidate Central Metric

This paper introduces a new metric that may be a better north- star for governed self-improvement than benchmark score or commit velocity. **Operator Intervention Half-Life (OIH)** is defined as:

> *The wall-clock time required for a repeated human- intervention class to become detected, patched by the loop, gated, and retired such that the next occurrence is handled autonomously.*

The motivation is direct. A benchmark score measures *what an agent can solve.* A commit count measures *how much the loop is producing.* Neither directly measures the property the metric is intended to capture for governed self-improvement, which is: *how quickly does the operator's intervention budget for any given failure class go to zero?*

6.7.1 Operationalization

For a given intervention class C (e.g., "cherry-pick conflict recovery", "PRD lifecycle dual-directory cleanup", "backend rate-limit failover"), OIH(C) is computed as:

1. **Detection event.** First timestamp at which the human operator manually performed an intervention of class C during the reporting window.
2. **Patch event.** First commit timestamp at which a patch landed on `main` whose intent (per commit message classification) was to make class C handled by the loop.
3. **Gate event.** First subsequent reboot of the supervised runner under the new substrate.
4. **Retirement event.** First subsequent occurrence of class C in the runtime that completed without the operator's shepherd skill firing a Tier-A escalation for C.

OIH(C) is the wall-clock duration from (1) to (4). If (4) never occurs in the reporting window, OIH(C) is right-censored at the window's end. This paper adopts a survival-analysis framing deliberately: the central question is *what fraction of intervention classes have been retired by time t*, not just the mean retirement time of the retired ones.

6.7.2 Caveats

OIH is harder to measure than commit count. The operator must log every intervention with a class label, which I have done only partially during the reporting window. The retrospective classification of feedback memory entries (see Appendix D and the per-incident traces in ğ7.4) gives a rough lower bound on operator-intervention counts but does not give a clean OIH curve. This paper names the metric here so that a v2 of the system can instrument it directly: every shepherd-skill Tier-A escalation should emit a structured event with the intervention class, the operator action taken, and a back-pointer to the candidate patch (if any) that retires the class. A follow-up paper should publish a proper OIH survival curve once that instrumentation is in place.

6.7.3 The orphaned-fix incident as a single OIH data point

One incident in the reporting window provides an unambiguous upper-bound data point for one intervention class. On 2026-05-21 the human operator authored a maintenance fix that was orphaned because the RSI loop had independently shipped a structurally-equivalent fix thirty minutes earlier. For that specific failure class, OIH $\leq$ 30 minutes. This paper claims no generalization from a single data point; the incident is offered as existence proof that the metric can in principle reach operationally-significant values within the reporting cadence of the loop.

6.8 Companion Metrics for Governed RSI

OIH measures the *time* dimension of operator displacement. Three additional metrics describe complementary dimensions of governed self-improvement; this paper names them here so that future work has a shared vocabulary. None of these metrics are intended to compete with OIH — they are companions to it, each surfacing a different facet of the loop's empirical behaviour.

6.8.1 Loop-Closure Latency (LCL)

> *The wall-clock time from the **first** observed occurrence of a recurring failure trace to the **merged commit** of a detector or recovery patch that retires the class.*

OIH measures end-to-end displacement of an *operator* intervention. LCL measures end-to-end displacement of a *failure* — even when the failure was never bad enough to trigger operator attention. LCL is therefore a finer-grained signal than OIH; a healthy loop can have LCL well below OIH, because most failures are retired by the loop before they escalate to the operator. The two metrics together form a two-tier funnel: failures whose LCL exceeds a budget become candidates for operator intervention, and the OIH of that intervention class then measures how quickly the system retires the *operator-visible* version.

LCL is operationalized via the trace harvester's `first_seen_at` timestamps (per cluster) and the audit-chain record of the merged patch that the clusterer subsequently stops emitting for that cluster. Like OIH, full LCL instrumentation is a v2 deliverable; the metric is named here to anchor future telemetry work.

6.8.2 Operator Displacement Ratio (ODR)

> *The fraction of operator-proposed fixes that were already shipped or preempted by RSI before the operator's commit could reach* `origin/main`.

ODR is the most direct measure of whether the loop is ahead of or behind the operator. The orphaned-fix incident of 2026-05-21 (ğ7.4) contributes one event to the numerator. A v2 instrumentation would mark every operator commit that was rendered empty by a structurally-equivalent prior loop commit, producing a running ODR over the reporting window.

6.8.3 Gate Sharpness (GS)

> *The fraction of promotion-gate decisions whose composite score lies in the boundary band* $[\,-\,,\,+\,]$*, where* *is a small neighborhood around the accept threshold .*

A healthy gate should make most of its decisions *away from* the boundary — high composites accept cleanly, low composites reject cleanly. A high-GS gate (lots of boundary decisions) suggests either that candidate quality is bunched at the boundary or that the threshold is poorly calibrated. This paper reports GS as a sanity-check on Table 4's verdict distribution rather than as a primary metric.

6.8.4 Self-Improvement Mix (SIM)

> *The ratio of structural RSI commits (changes to the RSI machinery itself) to* `[auto]` *worker-output commits (application-facing fixes shipped by the loop).*

In the current first-parent `origin/main` history, the cleanly reproducible side of this metric is the numerator: 1,794 commits match structural RSI/shepherd prefixes. The older `[auto]` worker-output denominator used in earlier drafts is not reproducible from the current first-parent subject lines, so this paper no longer uses SIM as a headline result. The metric remains useful, but it requires a committed attribution export that separates application-facing worker output from analysis refreshes, skill projections, and manual operator commits.

---

7. Experimental Setup

7.1 Hardware

- **Host**: Linux workstation, 6.17 kernel, Conda environment `Datakult_2.1`

- **GPU**: RTX PRO 6000 Blackwell, 96 GB VRAM
- **LLM serving**: OpenAI Codex / Responses routes using `gpt-5.5` with `xhigh` reasoning for Codex App Server worker dispatch and `/shepherd` operation; local vLLM through `local-vllm-primary` (`gpt-oss-120b`) for judge and analysis roles.
- **Worker backend**: `codex_app_server` primary for RSI worker dispatch, using the repo Codex/OpenAI default (`gpt-5.5`, `xhigh`); the shepherd uses the same high-capability route. Claude CLI remains an outer failover/backend option.

7.2 Software stack

- **LLM Adapter**: All LLM calls routed through `Datakult_AGI_Utils/llm_adapter/`. Direct provider calls are forbidden by lint.
- **Redis**: Centralized via `Redis_Master_Manager_Client.py`.
- **Memory**: Mem0 via `Mem0_Master_Manager_Client.py`.
- **Task schema**: `UnifiedTaskSchema` for all task operations.
- **Prompts**: `Master_Prompts/` with no inline strings.

7.3 Runtime topology

Three lanes run in parallel during a single campaign iteration:

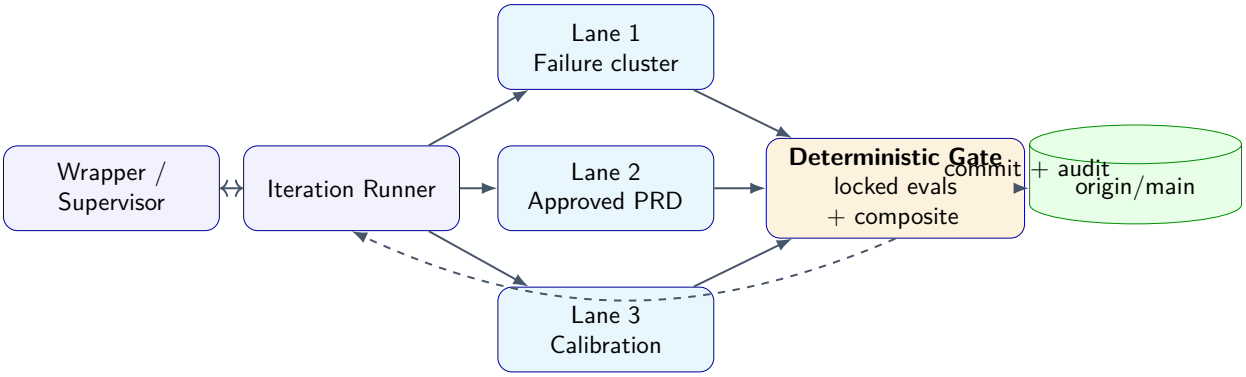

Figure 4: Single-iteration runtime topology. Three lanes operate concurrently. The promotion gate is the single serialization point; lanes are otherwise independent. The 25-iteration cap on campaign length is a hygiene mechanism: at the cap, the supervisor archives the campaign's state, harvests a fresh trace bundle, and starts a new campaign.

7.4 Operator role

The human operator runs the `/shepherd` skill on a 10-minute loop. The shepherd uses the same high-capability OpenAI route as the Codex App Server worker path (`gpt-5.5`, `xhigh`); local `gpt-oss-120b` is reserved for judge and analysis roles. The shepherd inspects heartbeats, audits the promotion chain, and intervenes only for *f[Truncated]*