Telemetry-Driven Self-Improvement: Structured Learning Loops for Autonomous Coding Agents

Hanzo AI Research
Hanzo AI Inc (Techstars '17), Los Angeles, CA
research@hanzo.ai

February 2026

Abstract

We present a **4-Loop Self-Improvement Architecture** for autonomous coding agents that replaces aspirational narrative self-reflection with structured, telemetry-driven optimization. The core thesis is that immutable structured data about tool invocations—gathered continuously and silently—constitutes a richer and more reliable learning signal than LLM-generated summaries of past performance. The four loops operate at distinct timescales: Loop 0 captures every tool invocation (always-on); Loop 1 executes the *Build-It-Now* protocol to eliminate recurring friction within the same session ($\leq$5 min); Loop 2 fires on structural correction signals to encode user preferences in real-time ($\leq$30 sec); and Loop 3 synthesizes session-level learnings via constrained templates ($\leq$2 min). Loop 4 provides a human-gated cross-session maintenance pass that aggregates evidence before proposing persistent changes.

We formalize the self-improvement problem as optimization over tool success rates, prove convergence of the trigger-based update rule under mild regularity conditions, and establish an information-theoretic bound showing that append-only telemetry logs preserve $\Omega(\log n)$ times more decision-relevant entropy than equivalently-sized narrative summaries. Deterministic triggers outperform LLM self-judgment in both precision (0.94 vs. 0.61) and recall (0.89 vs. 0.72) for identifying improvement opportunities, as measured against a ground truth of repeated-failure patterns. Our architecture extends and complements bmo [bmo, 2024], ngrok's self-improving agent, and is consistent with the empirical finding that 11 targeted tool improvements over 100 sessions yield measurable productivity gains. The implementation is available as a TypeScript plugin for the Hanzo bot SDK.

Contents

1 Introduction

The promise of autonomous coding agents that improve with use has attracted substantial research attention [Shinn et al., 2023, Wang et al., 2023, 2024, Wu et al., 2023]. The intuition is appealing: an agent that can identify its own weaknesses and fix them should, over time, require less human supervision and handle more complex tasks. Yet in practice, deployed agents rarely exhibit reliable self-improvement, and the mechanisms proposed in the literature frequently fail to transfer to production settings.

We argue that most existing approaches share a fundamental design flaw: they rely on LLMs to accurately evaluate their own performance and decide when improvement is warranted. This creates a circularity problem. The same model whose behavior we wish to improve is asked to judge whether that behavior is deficient—and to do so using the same context window that the deficient behavior just consumed.

The Context Window Problem. A typical coding session accumulates thousands of tokens of interaction history. After a long session, the agent's context window contains a dense mixture of task descriptions, tool outputs, intermediate reasoning, code diffs, error messages, and conversational repairs. When asked to reflect on "what went wrong," the LLM must process this entire narrative to produce a summary. Information theory tells us this is lossy: any fixed-length summary of an arbitrary-length sequence discards information. More practically, the patterns most relevant to self-improvement—repeated tool failures, systematic error modes, recurring user corrections—are precisely the patterns that are most diluted by narrative summaries because they are distributed across many non-contiguous tokens.

The Self-Evaluation Problem. Beyond lossy summarization, LLMs are systematically miscalibrated self-evaluators during task execution [Guo et al., 2017, Kadavath et al., 2022]. An agent engaged in a complex coding task allocates its attention to the task, not to meta-level monitoring. When subsequently prompted to reflect, it exhibits well-documented biases: recency bias (overweighting the last few turns), sycophancy (avoiding negative self-assessments that conflict with task completion), and confabulation (generating plausible-sounding but inaccurate retrospectives).

The Telemetry Thesis. We propose a different foundation. Every tool invocation produces a small, structured, machine-readable record: which tool was called, with what arguments, what the outcome was, and how long it took. These records are appended to a JSONL log independently of the agent's context window. They are never summarized, never compressed by the LLM, and never lost to attention decay. They accumulate silently and are available for deterministic analysis at any time. We call this the *telemetry thesis*: immutable structured data about tool invocations is a superior substrate for self-improvement than narrative summaries of those invocations.

The Build-It-Now Protocol. A second failure mode of existing systems is deferral. When an agent notices a recurring problem, it has two choices: fix it now, or record it for later. Every system that records problems for later produces growing backlogs of `OPPORTUNITIES.md` files that are never acted upon. We propose instead the *Build-It-Now* (BIN) protocol: when a deterministic trigger fires (three identical failure modes in one session), the agent pauses its current task, builds a targeted micro-tool to eliminate the friction, tests it, and resumes. No backlog. No deferral. No aspirational documentation.

Contributions. This paper makes the following contributions:

1. **4-Loop Architecture:** A formal description of four self-improvement loops operating at distinct timescales, with precise trigger conditions, latency bounds, and interaction semantics (Section 3).
2. **Build-It-Now Protocol:** An anti-deferral mechanism with a formal algorithm (FRICTION-DETECT) and a multi-stage build pipeline that guarantees atomic rollback on failure (Section 5).
3. **Formal Analysis:** Convergence guarantees for the trigger-based update rule, an information-theoretic bound on telemetry entropy vs. narrative entropy, and a precision/recall analysis of deterministic vs. LLM-based trigger detection (Section 9).
4. **Anti-Pattern Catalog:** A systematic enumeration of the design anti-patterns that the 4-Loop Architecture is engineered to prevent (Section 10).
5. **Implementation:** A production TypeScript implementation as a Hanzo bot plugin with full telemetry service, friction detector, and active learning hooks (Section 11).

Paper Outline. Section 2 reviews related work. Section 3 presents the 4-Loop Architecture. Section 4 through Section 8 detail each loop. Section 9 provides formal analysis. Section 10 catalogs anti-patterns. Section 11 describes the implementation. Section 12 presents evaluation results. Section 13 discusses future directions.

2 Related Work

2.1 bmo: ngrok's Self-Improving Agent

The most directly related work is bmo [bmo, 2024], ngrok's autonomous coding agent with a self-improvement loop. bmo maintains a library of tools and a memory of past interactions, and uses LLM-based reflection to identify opportunities to build new tools. Over 100 sessions, bmo built 11 tools that measurably reduced friction in repeated tasks. bmo's key insight—that agents should build tools to address their own limitations—directly inspires our Loop 1 design.

However, bmo's improvement mechanism has two limitations our architecture addresses. First, bmo uses LLM judgment to decide when to build new tools, which suffers from the self-evaluation problem described in Section 1. Second, bmo's tool-building is session-agnostic; there is no structured mechanism to aggregate evidence across sessions before investing in a new tool. Our architecture addresses both: deterministic triggers replace LLM judgment, and Loop 4's maintenance pass aggregates cross-session evidence.

2.2 Reflexion

Shinn et al. [2023] introduced Reflexion, a framework in which agents maintain a "self-reflection" string that is prepended to the context on subsequent attempts at a failed task. Reflexion demonstrates that verbal reflection improves task performance across a range of benchmarks. Our work is complementary but takes a different design stance: where Reflexion is optimized for single-task retry scenarios (try, reflect, retry), our architecture targets persistent skill accumulation across different tasks over many sessions.

More importantly, Reflexion's verbal reflection is stored as LLM-generated text, which inherits the entropy limitations we formalize in Section 9.3. We show that structured telemetry records preserve significantly more decision-relevant information per byte.

2.3 Voyager

Wang et al. [2023] presented Voyager, an LLM-powered agent in Minecraft that builds a skill library via automated curriculum learning. Voyager's "skill library" is the closest precedent to our Loop 1 tool-building mechanism. Key differences: Voyager operates in a static simulated environment where the space of skills is well-defined, while our architecture operates in open-ended software development environments where friction modes are not known in advance. Voyager uses an LLM to decide when to codify a skill; we use deterministic pattern matching over telemetry streams.

2.4 OpenHands

Wang et al. [2024] describes OpenHands (formerly OpenDevin), a platform for AI software engineers that emphasizes action space design and sandboxed execution. OpenHands provides the operational substrate on which our architecture can be layered—it handles low-level execution, while we handle the meta-level improvement loop. OpenHands does not itself provide a self-improvement mechanism beyond standard context management.

2.5 AutoGen

Wu et al. [2023] introduced AutoGen, a multi-agent conversation framework that enables complex agent orchestration via structured dialogue. AutoGen's conversational model allows agents to request help from specialist agents, which is one mechanism for ad-hoc improvement. However, AutoGen is designed for multi-agent coordination rather than within-agent self-improvement, and does not address the telemetry or anti-deferral problems we study here.

2.6 Self-Refine

Madaan et al. [2023] demonstrated that LLMs can improve their outputs through iterative self-critique and refinement within a single inference pass. Self-Refine is the most prominent instantiation of the "LLM as self-evaluator" paradigm that we critically examine. While Self-Refine works well for short tasks with clear quality signals (code compilation, math verification), it degrades on long-horizon tasks where quality signals are delayed and the LLM's self-critique is uncalibrated. Our formal analysis in Section 9.4 quantifies this degradation.

2.7 Tool-Use and Agent Skill Acquisition

More broadly, the problem of skill acquisition in agents has been studied in the context of tool-use [Schick et al., 2024, Qin et al., 2024], program synthesis [Chen et al., 2021], and reinforcement learning [Sutton and Barto, 2018]. Our work is distinguished by its focus on the *trigger mechanism*—not just how to build new skills, but when to build them—and its insistence on deterministic triggers over learned policies for production reliability.

3 The 4-Loop Architecture

3.1 Overview

We define a *self-improving coding agent* as an agent that operates in a software development environment and exhibits monotonically non-decreasing expected task success rate as the number of completed sessions increases. The 4-Loop Architecture provides a concrete instantiation of this definition via four feedback loops operating at distinct timescales.

Definition 1 (4-Loop Self-Improvement System)**.** *A 4-Loop Self-Improvement System* $\mathcal{S} = (\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$ *consists of five loops:*

- $\mathcal{L}_0$*: The* Telemetry Pipeline*, operating continuously with zero latency overhead.*
- $\mathcal{L}_1$*: The* Build-It-Now Loop*, event-driven with latency* ≤ 5 *minutes.*
- $\mathcal{L}_2$*: The* Active Learning Loop*, hook-driven with latency* ≤ 30 *seconds.*
- $\mathcal{L}_3$*: The* Session Reflection Loop*, end-of-session with latency* ≤ 2 *minutes.*
- $\mathcal{L}_4$*: The* Maintenance Pass*, every* N *sessions, human-gated with no latency bound.*

Figure 1 illustrates the loop structure and information flows. The critical invariant is that no loop writes to the agent's active context window. Each loop reads from and writes to persistent stores (telemetry log, learned facts database, tool library, proposal queue) that are loaded at session start, not accumulated during execution.

4-Loop Self-Improvement Architecture

Loop	Trigger	Latency	Reads	Writes
$\mathcal{L}_0$	Every tool call	0 ms	–	Telemetry log
$\mathcal{L}_1$	$\geq$3 identical failures	$\leq$5 min	Telemetry log	Tool library
$\mathcal{L}_2$	Correction signal	$\leq$30 sec	Context diff	Facts DB
$\mathcal{L}_3$	Session end	$\leq$2 min	Telemetry log	Reflection store
$\mathcal{L}_4$	Every N sessions	Unbounded	All stores	Proposal queue

Figure 1: Summary of the 4-Loop Architecture. Each loop operates independently with a well-defined trigger condition, latency bound, and storage contract. No loop pollutes the active context window.

3.2 State Space

The agent state at session s is a tuple:

$$\Sigma^{(s)} = \big(\mathcal{T}^{(s)},\ \mathcal{F}^{(s)},\ \mathcal{K}^{(s)},\ \mathcal{R}^{(s)},\ \mathcal{P}^{(s)}\big), \tag{1}$$

where $\mathcal{T}^{(s)}$ is the telemetry log (all sessions up to s), $\mathcal{F}^{(s)}$ is the learned facts database, $\mathcal{K}^{(s)}$ is the tool library (built-in plus agent-constructed), $\mathcal{R}^{(s)}$ is the reflection store, and $\mathcal{P}^{(s)}$ is the human-pending proposal queue.

The self-improvement objective is to maximize expected task success rate $\mu^{(s)} = \mathbb{E}[\text{success}(t) \mid \Sigma^{(s)}]$ over the task distribution. Each loop updates a subset of $\Sigma^{(s)}$, and the state is loaded fresh at the start of each session. This design ensures that improvements from one session are available in subsequent sessions without contaminating within-session context.

4 Loop 0: Telemetry Pipeline

4.1 Formal Definition

The telemetry pipeline is the foundation on which all other loops depend. Its design must satisfy three requirements: (1) zero latency overhead on the critical path, (2) append-only storage for tamper-evidence and crash safety, and (3) sufficient structural richness to support all downstream analyses.

Definition 2 (ToolInvocation Record)**.** *A* ToolInvocation *record τ is a tuple:*

$$\tau = \big(id,\ session\_id,\ ts,\ tool,\ args,\ outcome,\ duration\_ms,\ failure\_mode\big), \tag{2}$$

where:

- *$id \in \{0,1\}^{256}$: Globally unique record identifier (SHA-256 of content).*
- *$session\_id \in \mathbb{N}$: Monotonically increasing session counter.*
- *$ts \in \mathbb{R}_{>0}$: Unix timestamp with millisecond precision.*
- *$tool \in \mathcal{V}_T$: Tool identifier from the tool vocabulary.*
- *$args \in \mathcal{A}$: Serialized argument record (JSON, size-bounded).*
- *$outcome \in \{\text{SUCCESS}, \text{FAILURE}, \text{TIMEOUT}, \text{CANCELLED}\}$: Categorical outcome.*
- *$duration\_ms \in \mathbb{N}$: Wall-clock execution time.*
- *$failure\_mode \in \mathcal{V}_F \cup \{\perp\}$: Categorical failure mode or null on success.*

Remark 1. *The content-addressed identifier id enables deduplication and serves as an integrity check: any modification to a record is detectable by recomputing the hash. This property is important for Loop 4's cross-session aggregation, which must be robust to storage corruption.*

4.2 Storage Architecture

Records are appended to a JSONL (JSON Lines) file, one record per line. JSONL is preferred over binary formats for three reasons:

1. **Crash safety:** A partial write affects only the last line. All preceding lines remain valid and parseable.

2. **Tier portability:** The same file format works identically on a local laptop, a remote container, and an EC2 instance. NATS JetStream can replay from JSONL without schema migration.

3. **Zero write amplification:** Append-only writes never trigger compaction, B-tree rebalancing, or WAL flushing. The telemetry path adds at most one `write(2)` syscall per tool invocation.

Records are simultaneously published to a NATS JetStream subject `telemetry.session_id.tool` for real-time consumption by Loops 1 and 2. The NATS layer provides durability (disk-backed JetStream), replay capability (for recovering Loop 1 analysis after agent restart), and subject-based filtering (Loop 2 subscribes only to its trigger subjects).

Definition 3 (Telemetry Log)**.** *The telemetry log $\mathcal{T}^{(s)}$ at session s is the ordered sequence:*

$$\mathcal{T}^{(s)} = \big(\tau_1, \tau_2, \ldots, \tau_{|\mathcal{T}^{(s)}|}\big), \quad \tau_i.ts \le \tau_{i+1}.ts, \tag{3}$$

stored as an append-only JSONL file. The session slice *$\mathcal{T}_{cur}^{(s)}$ is the subsequence of records with $session\_id = s$.*

4.3 Failure Mode Taxonomy

The failure mode vocabulary $\mathcal{V}_F$ is a hierarchical taxonomy with three levels:

Table 1: Failure Mode Taxonomy (Level-1 categories with examples).

Code	Category	Example Instances
`PERM`	Permission error	File not writable, socket permission denied
`NOTFOUND`	Resource not found	File missing, URL 404, package not installed
`TIMEOUT`	Execution timeout	Command exceeded wall-clock limit
`SYNTAX`	Syntax/parse error	Malformed JSON argument, invalid regex
`RUNTIME`	Runtime exception	Uncaught exception, segfault, OOM
`TOOL_GAP`	Missing capability	No tool for requested operation
`ARGS`	Argument error	Wrong type, out of range, missing required field
`CONFLICT`	State conflict	Git merge conflict, lock contention
`NETWORK`	Network failure	DNS failure, connection refused, TLS error

Level-2 and Level-3 sub-codes refine each category. The full taxonomy is defined in the SDK and is extensible: new failure modes can be registered by tool authors without modifying the core taxonomy.

5 Loop 1: Build It Now

5.1 Motivation and Trigger Condition

The Build-It-Now (BIN) loop is the most novel contribution of this architecture. Its design philosophy is that *the best time to fix a recurring problem is the third time it occurs*, not at the end of the session, not in the next sprint, and not after it has been documented in a backlog. The third occurrence provides enough evidence that the problem is systematic (ruling out one-off errors) while the session context is still fresh enough to build a targeted solution.

Definition 4 (BIN Trigger)**.** *The BIN trigger $\mathcal{T}_{BIN}$ fires when the telemetry stream contains a pattern match:*

$$\mathcal{T}_{BIN}(\mathcal{T}_{cur}) = \mathbf{1}\big[\exists\,(t, f) \in \mathcal{V}_T \times \mathcal{V}_F : \#\{\tau \in \mathcal{T}_{cur} \mid \tau.tool = t,\ \tau.failure\_mode = f\} \ge \theta\big], \tag{4}$$

where $\theta = 3$ is the default threshold and $\mathcal{T}_{cur}$ is the current session slice.

The threshold $\theta = 3$ is motivated by the coupon-collector problem: with k distinct failure modes, the expected number of trials to observe any mode three times is $3k$ when modes are uniformly distributed, and substantially less when one mode is dominant (which is the case of interest). We show in Section 9.4 that $\theta \in \{2, 3, 4\}$ yields the best precision-recall tradeoff, with $\theta = 3$ being Pareto-optimal in most regimes.

5.2 FRICTION-DETECT Algorithm

Algorithm 1 specifies the FRICTION-DETECT procedure that continuously monitors the telemetry stream and fires the BIN trigger.

Algorithm 1 FRICTION-DETECT(S, θ)

```
Require: Telemetry stream $S$ (NATS subscription), threshold $\theta$
Ensure: BIN trigger events with $(t, f, \text{evidence})$ payload
$C \leftarrow$ DefaultDict($\mathbb{N}$)                ▷ Failure counter: $(t, f) \to \mathbb{N}$
$E \leftarrow$ DefaultDict(list)                        ▷ Evidence collector: $(t, f) \to [\tau]$
fired $\leftarrow \emptyset$                            ▷ Set of already-triggered pairs
for each record $\tau$ from $S$ do
    if $\tau$.outcome = FAILURE and $\tau$.failure_mode $\neq \bot$ then
        $k \leftarrow (\tau\text{.tool},\ \tau\text{.failure\_mode})$
        $C[k]$ += 1
        $E[k]$.APPEND($\tau$)
        if $C[k] \geq \theta$ and $k \notin$ fired then
            fired.ADD($k$)
            emit BINEvent($t = k_1,\ f = k_2,\ \text{evidence} = E[k]$)
        end if
    end if
end for
```

5.3 BIN Build Pipeline

When a BINEvent fires, the agent executes the following multi-stage pipeline. The pipeline is designed to be atomic: if any stage fails, the agent rolls back to its pre-BIN state and resumes the paused task.

1. **Pause.** Serialize the current task state (context snapshot, pending actions queue) to a checkpoint file.

2. **Design.** Prompt the LLM with: (a) the BINEvent evidence (up to 3 example failure records), (b) the current tool library manifest, and (c) a structured template requesting a micro-tool design. The design must specify: tool name, input schema, output schema, implementation sketch, and acceptance criteria.

3. **Build.** Generate tool implementation from the design. Tools are TypeScript modules conforming to the Hanzo bot plugin interface. The LLM generates the implementation; a static linter and type checker run immediately.

4. **Test.** Execute the acceptance criteria as unit tests against the reproduced failure case (reconstructed from the evidence records). A tool that does not pass its own acceptance criteria is not loaded.

5. **Gate.** If all acceptance criteria pass, the tool is added to the tool library with a `status: "probationary"` flag. A probationary tool requires two more successful uses before its status is promoted to `"stable"`.
6. **Reload.** Load the new tool into the active tool registry.
7. **Resume.** Restore the serialized task state and continue execution with the new tool available.

Algorithm 2 BIN-BUILD-PIPELINE(event, $\mathcal{K}$)

```
Require: BINEvent event, current tool library $\mathcal{K}$
Ensure: Updated tool library $\mathcal{K}'$ or $\mathcal{K}$ on failure
checkpoint ← SerializeContext()
design ← LLM-Design(event.evidence, $\mathcal{K}$)
if design = $\perp$ then
    return $\mathcal{K}$                    ▷ LLM could not produce design; resume without change
end if
impl ← LLM-Implement(design)
lint_ok ← TypeCheck(impl)
if ¬lint_ok then
    return $\mathcal{K}$
end if
test_result ← RunAcceptanceCriteria(impl, design.criteria, event.evidence)
if ¬test_result.passed then
    return $\mathcal{K}$
end if
tool ← Package(impl, status = "probationary")
$\mathcal{K}' \leftarrow \mathcal{K} \cup \{$tool$\}$
ReloadRegistry($\mathcal{K}'$)
RestoreContext(checkpoint)
return $\mathcal{K}'$
```

5.4 Anti-Deferral Properties

Proposition 1 (No Backlog Accumulation)**.** *In a system running the BIN protocol, the expected size of the improvement backlog grows as $O(\log s)$ in the number of sessions s, compared to $O(s)$ in deferral-based systems.*

Proof. In a deferral-based system, each session generates at most B improvement opportunities (where B is session length divided by minimum failure stride). These are appended to the backlog. Without a consumption mechanism of comparable rate, the backlog grows as $O(B \cdot s) = O(s)$.

In the BIN system, each failure pattern (t, f) triggers a build the first time it reaches threshold θ. Subsequent occurrences of the same (t, f) pair after a successful build do not trigger another build (the tool is already in the library). New (t, f) pairs can only arise from the vocabulary expansion rate, which grows at most as $O(\log s)$ by the information-theoretic lower bound on new information in a stationary environment (new failure modes are progressively rarer as the tool library matures). Therefore the backlog size is $O(\log s)$. □

6 Loop 2: Active Learning

6.1 Motivation

Users of coding agents routinely correct agent behavior through natural language. “No, use tabs not spaces.” “Actually, don’t use async/await here.” “Stop importing from lodash.” These corrections are high-quality preference signals: they are direct, unambiguous, and immediately actionable. The challenge is detecting them.

Crucially, these corrections occur within the conversational flow and are not labeled as corrections by the user. An LLM asked to identify corrections in a transcript will find them, but with high false-positive rate (flagging instructions as corrections) and meaningful false-negative rate (missing implicit corrections that come as rewrites rather than explicit statements). We propose a structural detection approach that is faster and more reliable.

6.2 Structural Signal Detection

Definition 5 (Correction Signal)**.** *A correction signal σ is detected when any of the following structural conditions hold in the most recent $W = 5$ turns:*

- ***Edit-distance trigger:*** *The user’s latest message m_t has normalized Levenshtein distance $d_L(m_t, m_{t-2})/|m_{t-2}| < 0.3$, where m_{t-2} is the agent’s preceding response (indicating direct text editing of the agent’s output).*
- ***Negation trigger:*** *m_t begins with or contains one of the negation phrases in $\mathcal{N}$ (e.g., “no,” “actually,” “don’t,” “stop,” “instead”).*
- ***Tool override trigger:*** *The user explicitly invokes a tool with parameters that directly contradict the agent’s last tool call on the same target.*
- ***Rewrite trigger:*** *The user provides a full rewrite of agent-generated code that is $> 50\%$ different by line count.*

6.3 LearnedFact Schema

When a correction signal σ is detected, the active learning loop extracts a LearnedFact record and appends it to the facts database.

Definition 6 (LearnedFact Record)**.** *A* LearnedFact *record ϕ is a tuple:*

$$\phi = \big(id,\ content,\ confidence,\ scope,\ source,\ ts\big), \tag{5}$$

where:

- *$content \in \Sigma^*$: Natural language statement of the fact.*
- *$confidence \in [0, 1]$: Prior confidence in the fact.*
- *$scope \in \{global, project, session\}$: Applicability scope.*
- *$source \in \{user\text{-}stated, inferred, corrected\}$: Provenance.*

A critical design decision is that user-stated facts—facts extracted from a negation trigger where the user explicitly states a preference—receive confidence = 1.0 and scope = project by default. This reflects the epistemic status of direct user instruction: it is not a hypothesis to be tested but a ground truth to be respected.

Inferred facts (extracted from edit-distance or rewrite triggers) receive confidence = 0.7 initially, which is updated by subsequent confirmations or contradictions. A conflicting user-stated fact always supersedes an inferred fact, regardless of confidence.

6.4 Why Hooks Beat LLM Judgment

Proposition 2 (Structural Detection Superiority)**.** *For correction detection in coding agent sessions, structural signal detection achieves precision $P_s \geq P_{LLM}$ and recall $R_s \geq R_{LLM}$ with time complexity $O(1)$ per turn compared to $O(n)$ for LLM-based detection (where n is context length).*

The argument is empirical: the structural triggers in Definition 5 correspond directly to the linguistic behaviors users exhibit when correcting agents. The negation trigger alone captures $> 80\%$ of explicit corrections. The remaining triggers capture implicit corrections that LLMs systematically miss because they require comparing tool call arguments to prior tool call arguments—a structured comparison that LLMs perform unreliably but code performs exactly.

7 Loop 3: Session Reflection

7.1 Motivation and Design

Session reflection occupies the final 2 minutes of each session. Its purpose is to synthesize the session's telemetry and context into a compact structured record that Loop 4 can aggregate across sessions. The key design constraint is that reflection must use *constrained templates* rather than open-ended prompts.

Open-ended reflection prompts ("What did you learn this session?") produce outputs that are useful as prose but difficult to aggregate. Two sessions with similar issues will produce different prose descriptions of those issues, making cross-session pattern detection unreliable. Constrained templates with fixed output schemas solve this.

7.2 Reflection Template

The session reflection uses the following three-question template, which the LLM answers with structured JSON output:

Question 1: What worked? Enumerate tool invocations or strategies with outcome = SUCCESS that were reused ≥ 2 times in the session. Report as: {`tool`, `strategy`, `success_count`, `why_effective`}.
Question 2: What failed repeatedly? Enumerate (t, f) pairs with count ≥ 2 that did not trigger the BIN loop (i.e., threshold not yet reached). Report as: {`tool`, `failure_mode`, `count`, `hypothesis`}.
Question 3: What should change? Based on the above, enumerate proposed changes to the tool library, system prompt, or behavior patterns. Each proposal must reference specific evidence from Questions 1 and 2. No proposal without evidence is recorded.

Definition 7 (SessionReflection Record)**.** *A* SessionReflection *record ρ is a tuple:*

$$\rho = \big(\mathit{session\_id},\ \mathit{successes},\ \mathit{failures},\ \mathit{proposals},\ \mathit{telemetry\_digest}\big), \tag{6}$$

where telemetry_digest is a compact statistical summary of $\mathcal{T}^{(s)}_{cur}$ (tool usage histogram, failure rate per tool, session duration, task count) that is appended to the reflection record for Loop 4 consumption without requiring Loop 4 to replay the full log.

7.3 Constrained Templates vs. Open-Ended Reflection

Theorem 3 (Template Aggregation Advantage)**.** *Let R_T be the set of reflections produced by the constrained template, and R_O be the set of reflections produced by an open-ended prompt, over N sessions with the same underlying pattern distribution. Then the expected precision of cross-session pattern detection is:*

$$P(detect \mid R_T) \geq P(detect \mid R_O) + \Delta, \tag{7}$$

where $\Delta > 0$ depends on the entropy of the natural language description of patterns and the schema alignment between sessions.

Proof sketch. Cross-session pattern detection requires matching descriptions of similar events across sessions. For templated reflections, matching reduces to structured key equality (same tool name, same failure mode code). For open-ended reflections, matching requires semantic similarity, which has non-trivial false-negative rate $\varepsilon > 0$ (e.g., "permission denied on file writes" and "cannot write to read-only path" describe the same pattern but are lexically dissimilar). Therefore $P(\text{detect} \mid R_T) = 1 - O(|\mathcal{V}_T|^{-1})$ while $P(\text{detect} \mid R_O) \leq 1 - \varepsilon$ for some $\varepsilon > 0$ that depends on the natural language variability of failure descriptions. □

8 Loop 4: Maintenance Pass

8.1 Design Principles

Loop 4 is deliberately human-gated. This is not a limitation but a design choice based on three considerations:

1. **Risk asymmetry.** Changes to the base tool library or system prompt affect all future sessions. A false positive (deploying a change that degrades performance) is harder to recover from than a false negative (delaying a beneficial change by one cycle).
2. **Evidence requirement.** Loop 4 requires evidence from $N \geq 5$ sessions before generating a proposal. This ensures that proposals are based on systematic patterns, not one-off observations.
3. **Human expertise.** Changes to the tool library may require domain expertise that the LLM does not have (e.g., choosing between two correct implementations based on performance characteristics that cannot be measured in unit tests).

8.2 Cross-Session Aggregation

Loop 4 aggregates telemetry across all sessions to identify systematic patterns. The aggregation operates on the telemetry digest in each SessionReflection record, not the raw telemetry log, to bound processing time.

Definition 8 (MaintenanceProposal)**.** *A* MaintenanceProposal *record π is a tuple:*

$$\pi = \big(id,\ type,\ description,\ evidence,\ expected\_impact,\ risk,\ status\big), \tag{8}$$

where:

- *type* $\in \{$*new-tool, modify-tool, retire-tool, system-prompt, fact-update*$\}$
- *evidence: References to SessionReflection records and telemetry patterns supporting the proposal.*
- *expected_impact: Estimated change in tool success rate (with confidence interval).*
- *risk* $\in \{$*low, medium, high*$\}$*: Human-assessed risk category.*
- *status* $\in \{$*pending, approved, rejected, implemented*$\}$

8.3 Proposal Generation Algorithm

Algorithm 3 MAINTENANCE-PASS($\mathcal{R}_{1:N}, \mathcal{T}_{1:N}$)

```
Require: Session reflections $\mathcal{R}_{1:N}$, telemetry logs $\mathcal{T}_{1:N}$
Ensure: Set of MaintenanceProposal records
proposals $\leftarrow \emptyset$
patterns $\leftarrow$ AggregateFailurePatterns($\mathcal{T}_{1:N}$)
for each pattern $p = (t, f, \text{count}, \text{sessions}) \in$ patterns do
    if $p$.count $\geq \gamma_{\text{count}}$ and $|\text{sessions}| \geq \gamma_{\text{sessions}}$ then
        supporting $\leftarrow \{r \in \mathcal{R}_{1:N} \mid p \in r.\text{failures}\}$
        $\pi \leftarrow$ LLM-Propose($p$, supporting, current-toolkit)
        $\pi$.evidence $\leftarrow$ supporting
        $\pi$.status $\leftarrow$ "pending"
        proposals $\cup= \{\pi\}$
    end if
end for
for each proposal $\pi \in$ proposals do
    HumanReview($\pi$)                    ▷ Blocking: awaits human approval/rejection
    if $\pi$.status = "approved" then
        ImplementProposal($\pi$)
    end if
end for
return proposals
```

The thresholds $\gamma_{\text{count}} = 10$ and $\gamma_{\text{sessions}} = 3$ are defaults that balance sensitivity (detecting real problems) with specificity (avoiding spurious proposals from noisy sessions). These thresholds should be tuned to the agent's deployment context.

9 Formal Analysis

9.1 Self-Improvement as Optimization

We formalize the self-improvement objective in terms of tool success rates.

Definition 9 (Tool Success Rate). *The success rate $\mu_t^{(s)}$ of tool t at session s is:*

$$\mu_t^{(s)} = \mathbb{E}\big[\mathbf{1}[\tau.\mathit{outcome} = \textsc{success}] \mid \tau.\mathit{tool} = t,\ \Sigma^{(s)}\big], \tag{9}$$

where the expectation is over the task distribution conditional on the agent state at session s.

Definition 10 (Self-Improvement Objective). *The self-improvement objective is:*

$$\max_{\{\mathcal{K}^{(s)}, \mathcal{F}^{(s)}\}_{s \geq 1}} \lim_{S \to \infty} \frac{1}{S} \sum_{s=1}^{S} \mathbb{E}\left[\frac{1}{|\mathcal{V}_T|} \sum_{t \in \mathcal{V}_T} \mu_t^{(s)}\right], \tag{10}$$

i.e., maximize the long-run average tool success rate over all tools.

9.2 Convergence of the 4-Loop System

Theorem 4 (4-Loop Convergence)**.** *Under the following conditions:*

(A1) *The task distribution $\mathcal{D}$ is stationary.*

(A2) *The BIN build pipeline succeeds with probability $p_b > 0$ on any triggered (t, f) pair.*

(A3) *The space of failure modes $\mathcal{V}_F$ is finite.*

(A4) *The agent does not regress: adding a new tool does not decrease the success rate of existing tools.*

the 4-Loop System achieves:

$$\lim_{S\to\infty} \frac{1}{|\mathcal{V}_T|} \sum_{t\in\mathcal{V}_T} \mu_t^{(S)} = 1 \quad \textit{almost surely.} \tag{11}$$

Proof. We proceed by showing that every (t, f) pair with positive occurrence probability is eventually eliminated.

Step 1. Fix any (t, f) with positive probability $p_{tf} > 0$. Under stationarity (A1), the expected number of sessions until $\theta = 3$ failures of (t, f) occur in a single session is finite: $\mathbb{E}[\tau_{tf}] \le 1/p_{tf}^{\theta} < \infty$.

Step 2. When the BIN trigger fires for (t, f), the build pipeline succeeds with probability $p_b > 0$ (A2). If it fails, the same session continues accumulating failures, so the trigger fires again in the next session by Step 1. By the second Borel-Cantelli lemma, the build pipeline succeeds infinitely often, so with probability 1, there exists a session s^*_{tf} after which the (t, f) pair is eliminated from the active failure modes.

Step 3. By finiteness of $\mathcal{V}_F$ (A3), the number of distinct (t, f) pairs is $|\mathcal{V}_T| \times |\mathcal{V}_F| < \infty$. Therefore $s^* = \max_{t,f} s^*_{tf} < \infty$ with probability 1.

Step 4. By the no-regression assumption (A4), the tool library $\mathcal{K}^{(s)}$ is monotonically improving. After session s^*, all failure modes have been addressed and $\mu_t^{(s)} \to 1$ for all t as $s \to \infty$. □

Remark 2. *Assumption (A4) is the strongest assumption and may fail in practice if new tools have bugs. The probationary status mechanism in the BIN pipeline (Section 5) is designed to mitigate this by quarantining unproven tools until they accumulate positive evidence.*

9.3 Information-Theoretic Analysis

We now show formally that append-only telemetry logs preserve more decision-relevant information than narrative summaries.

Definition 11 (Decision-Relevant Information)**.** *Given telemetry log $\mathcal{T}$, the decision-relevant information for the BIN trigger is:*

$$I_{BIN}(\mathcal{T}) = H\big(\{(t, f) : \#_{tf}(\mathcal{T}) \ge \theta\}\big), \tag{12}$$

where H denotes Shannon entropy and $\#_{tf}(\mathcal{T})$ is the count of (t, f) failures in $\mathcal{T}$.

Theorem 5 (Telemetry Entropy Bound)**.** *Let $\mathcal{T}_n$ be a telemetry log of n records and let $\hat{\mathcal{T}}_k$ be any narrative summary of $\mathcal{T}_n$ using k tokens ($k \ll n$). Then:*

$$I_{BIN}(\mathcal{T}_n) \ge I_{BIN}(\hat{\mathcal{T}}_k) + \Omega(\log n)\,, \tag{13}$$

with high probability over the distribution of narrative summaries.

Proof sketch. The BIN trigger requires exact counts $\#_{tf}(\mathcal{T}) \geq \theta$ for specific (t, f) pairs. Reconstructing exact counts from a k-token summary requires at least $\log_2(\max_{t,f} \#_{tf}) \approx \log_2(n)$ bits per (t, f) pair that is near the threshold. A k-token summary has at most $k \cdot \log_2 |\mathcal{V}|$ bits of total information. For $k \ll n$, this cannot represent all pairs at threshold-resolution, so information about near-threshold pairs is discarded. The direct telemetry log represents every record exactly, preserving $O(\log n)$ more bits of threshold-resolution information per near-threshold pair. □

9.4 Deterministic vs. LLM-Based Triggers

We analyze the precision and recall of two trigger mechanisms for the BIN loop: deterministic counting (our approach) vs. LLM-based self-judgment.

Definition 12 (Trigger Precision and Recall)**.** *For a trigger mechanism M and ground truth G (set of (t, f) pairs that genuinely benefit from a new tool):*

$$P(M) = \frac{|M \cap G|}{|M|}, \quad R(M) = \frac{|M \cap G|}{|G|}. \tag{14}$$

Proposition 6 (Deterministic Trigger Dominance)**.** *For $\theta = 3$ and a stationary failure mode distribution, the deterministic counting trigger M_D satisfies:*

$$P(M_D) \geq P(M_{LLM}) \quad \text{and} \quad R(M_D) \geq R(M_{LLM}), \tag{15}$$

with probability $\geq 1 - \delta$ for any $\delta > 0$ when $n \geq n_0(\delta)$.

The empirical support for this proposition comes from our evaluation (Section 12), where we measure $P(M_D) = 0.94$, $R(M_D) = 0.89$ against $P(M_{\text{LLM}}) = 0.61$, $R(M_{\text{LLM}}) = 0.72$. The LLM trigger has higher recall in low-count regimes (detecting problems with fewer than θ occurrences) but much lower precision (many false positives from LLM hallucination about problems that did not occur frequently).

Table 2: Threshold sensitivity analysis for BIN trigger at $\theta \in \{2, 3, 4, 5\}$.

θ	Precision	Recall	F1	Builds/Session
2	0.81	0.95	0.87	2.3
3	**0.94**	**0.89**	**0.91**	1.1
4	0.97	0.72	0.83	0.6
5	0.99	0.58	0.73	0.3
LLM	0.61	0.72	0.66	1.8

Table 2 confirms that $\theta = 3$ is Pareto-optimal: it achieves the highest F1 score while keeping the builds-per-session rate below 1.5 (a reasonable upper bound for the 5-minute latency budget per session).

10 Anti-Patterns and Design Decisions

The 4-Loop Architecture is as much defined by what it *does not do* as by what it does. We enumerate the anti-patterns it is specifically engineered to prevent.

Table 3: Anti-patterns prevented by the 4-Loop Architecture.

Anti-Pattern	Description	4-Loop Prevention
Backlog growth	`OPPORTUNITIES.md` accumulates improvement ideas that are never acted upon, growing without bound.	Loop 1 (BIN) acts on triggers immediately. Proposition 1 guarantees $O(\log s)$ backlog size.
Context pollution	Self-improvement reasoning bleeds into the active context window, consuming tokens and degrading task focus.	All loops write to external stores. Zero impact on active context.
LLM trigger hallucination	LLM-based self-judgment generates false positives (“I should build a tool for X” when X is not a recurring problem).	Loop 1 uses deterministic counting. $P = 0.94$ vs. LLM $P = 0.61$.
Over-generalization	Building a tool that is too general to be reliable, or that duplicates existing tools with a different interface.	BIN build pipeline requires evidence from specific (t, f) pairs. Gate step requires acceptance criteria pass. Probationary status limits damage.
Auto-deployed regression	Automatically deploying a new tool or system prompt change that degrades performance on other tasks.	Loop 4 is human-gated. Loop 1 tools start probationary. No change without evidence.
Lossy narrative memory	Summarizing session events into narrative text that discards threshold-relevant information.	Loop 0 appends structured records. Theorem 5 guarantees $\Omega(\log n)$ entropy advantage.
Preference drift	User corrections are not recorded, so the agent repeats corrected behaviors in later sessions.	Loop 2 captures all correction signals and writes to persistent facts database.
Cross-session amnesia	Each session starts fresh, discarding all learning from prior sessions.	State $\Sigma^{(s)}$ (facts, tools, reflections) is loaded at session start.
Delayed correction	User corrections are applied only to the current session and not persisted.	Loop 2 writes to the persistent facts database immediately on detection.
Evidence-free proposals	Loop 4 proposals generated without supporting evidence from multiple sessions.	Proposal generation algorithm (Algorithm 3) requires $\gamma_{\text{sessions}} \geq 3$ supporting sessions per proposal.

11 Implementation

11.1 TypeScript Plugin Architecture

The 4-Loop Architecture is implemented as a TypeScript plugin for the Hanzo bot SDK. The plugin architecture exposes extension points at each loop boundary, allowing downstream integrators to customize trigger thresholds, storage backends, and LLM prompts without modifying the core loop logic.

```
interface SelfImprovementPlugin {
  // Loop 0: Telemetry
  onToolInvocation(record: ToolInvocation): Promise<void>;

  // Loop 1: Build It Now
  onBINTrigger(event: BINEvent): Promise<ToolLibraryUpdate | null>;

  // Loop 2: Active Learning
  onCorrectionSignal(signal: CorrectionSignal): Promise<LearnedFact | null>;

  // Loop 3: Session Reflection
  onSessionEnd(session: SessionSummary): Promise<SessionReflection>;

  // Loop 4: Maintenance Pass (human-gated)
  generateMaintenanceProposals(
    reflections: SessionReflection[],
    telemetry: TelemetryAggregate
  ): Promise<MaintenanceProposal[]>;
}
```

11.2 Telemetry Service

The telemetry service is the lowest-level component. It intercepts all tool invocations via a proxy wrapper around the tool registry, records each invocation as a ToolInvocation record, appends to the JSONL log, and publishes to NATS JetStream.

```
class TelemetryService {
  private log: AppendOnlyLog<ToolInvocation>;
  private nats: NATSClient;
  private sessionId: number;

  async record(invocation: ToolInvocation): Promise<void> {
    const record = {
      ...invocation,
      id: sha256(JSON.stringify(invocation)),
      session_id: this.sessionId,
      ts: Date.now(),
    };
    await this.log.append(record);
    await this.nats.publish(
      ‘telemetry.${this.sessionId}.${invocation.tool}‘,
      record
    );
  }
}
```

11.3 Friction Detector

The friction detector subscribes to the NATS telemetry stream and maintains the in-memory counter C from Algorithm 1. It emits BINEvents when thresholds are crossed.

```
class FrictionDetector {
  private counts: Map<string, number> = new Map();
  private evidence: Map<string, ToolInvocation[]> = new Map();
  private fired: Set<string> = new Set();

  async processRecord(record: ToolInvocation): Promise<BINEvent | null> {
    if (record.outcome !== ’failure’ || !record.failure_mode) return null;

    const key = ‘${record.tool}:${record.failure_mode}‘;
    const count = (this.counts.get(key) ?? 0) + 1;
    this.counts.set(key, count);
    this.evidence.set(key, [
      ...(this.evidence.get(key) ?? []),
      record
    ]);

    if (count >= this.threshold && !this.fired.has(key)) {
      this.fired.add(key);
      return {
        tool: record.tool,
        failure_mode: record.failure_mode,
        evidence: this.evidence.get(key)!,
      };
    }
    return null;
  }
}
```

11.4 Active Learning Hooks

The active learning hooks are registered as message-level observers in the bot conversation loop. They examine each user turn for correction signals without blocking the conversation.

```
class ActiveLearningHook {
  private readonly NEGATION_PHRASES =
    [’no,’, "don’t", ’stop’, ’instead’, ’actually’,
     "that’s wrong", ’use X not Y’];
  async onUserMessage(
    message: string,
    prevAgentResponse: string,
    prevToolCall?: ToolCall
  ): Promise<LearnedFact | null> {
    const signal = this.detectCorrectionSignal(
      message, prevAgentResponse, prevToolCall
    );
    if (!signal) return null;

    return await this.extractFact(message, signal);
  }

  private detectCorrectionSignal(
    msg: string,
    prevResponse: string,
    prevToolCall?: ToolCall
  ): CorrectionSignal | null {
    // Negation trigger
    if (this.NEGATION_PHRASES.some(p =>
        msg.toLowerCase().startsWith(p))) {
      return { type: 'negation', confidence: 1.0 };
    }
    // Edit-distance trigger
    const dist = levenshtein(msg, prevResponse);
    if (dist / prevResponse.length < 0.3) {
      return { type: 'edit', confidence: 0.7 };
    }
    // Tool override trigger
    if (prevToolCall && this.isOverride(msg, prevToolCall)) {
      return { type: 'tool_override', confidence: 0.9 };
    }
    return null;
  }
}
```

11.5 Reflection Hooks

The reflection hook fires at session end and uses the structured template from Section 7 to produce a SessionReflection record.

```
class ReflectionHook {
  async onSessionEnd(
    session: SessionContext,
    telemetry: ToolInvocation[]
  ): Promise<SessionReflection> {
    const digest = this.computeDigest(telemetry);
    const prompt = this.buildReflectionPrompt(telemetry, digest);
    const response = await this.llm.complete(prompt, {
      responseSchema: SessionReflectionSchema, // JSON Schema
      maxTokens: 512,
      temperature: 0.2, // Low temperature for structured output
    });

    return {
      session_id: session.id,
      successes: response.successes,
      failures: response.failures,
      proposals: response.proposals.filter(p => p.evidence.length > 0),
      telemetry_digest: digest,
    };
  }
}
```

11.6 Maintenance Tool and Proposal Lifecycle

The maintenance tool implements Algorithm 3 and exposes a CLI interface for human review. Proposals are stored as JSONL and presented with evidence summaries that reference specific session IDs and telemetry records.

```
$ hanzo-bot maintenance --sessions 20 --since 2026-01-01
Analyzing 20 sessions across 847 tool invocations...

Proposal 1 (HIGH CONFIDENCE): New tool for ’bash:TIMEOUT’
  - 34 failures across 12 sessions (avg 2.8/session)
  - Pattern: commands exceeding 30s default timeout
  - Proposed: background_exec tool with configurable timeout
  - Evidence: sessions 44, 51, 53, 57, 60, 61, ...
  [A]pprove  [R]eject  [S]kip  [V]iew evidence? A

Implementing...done. Tool added to library.
```

12 Evaluation

12.1 Evaluation Design

Our evaluation addresses three questions:

1. **Trigger quality:** How do deterministic triggers compare to LLM-based triggers in precision, recall, and F1?
2. **Improvement trajectory:** How does tool success rate evolve over sessions under the 4-Loop system?
3. **Loop interaction:** Do the four loops interact correctly, and does the anti-deferral property hold empirically?

12.2 Trigger Quality

We evaluate trigger mechanisms on a ground truth dataset of 500 simulated sessions with known injected failure patterns (50 unique (t, f) pairs with varying injection rates). We compare:

- $M_D(\theta = 3)$: Deterministic counting at threshold 3 (our approach)
- M_{LLM}: LLM-based self-judgment using a strong frontier model
- $M_D(\theta = 2)$, $M_D(\theta = 4)$: Ablation thresholds

Results are reported in Table 2. $M_D(\theta = 3)$ achieves F1 = 0.91 vs. M_{LLM} F1 = 0.66. The LLM's poor precision (0.61) reflects hallucinated trigger events: the LLM identifies improvement opportunities that are not supported by the failure count data, wasting build budget on problems that only occurred once or twice. The LLM's recall (0.72) is lower than $M_D(\theta = 3)$ (0.89), partly because LLM-generated analysis misses cases where the failure mode description in the context does not match the LLM's expected description.

12.3 Improvement Trajectory

We model the improvement trajectory analytically using the convergence result from Theorem 4. The expected time to eliminate a failure mode (t, f) with occurrence probability p_{tf} per session is:

$$\mathbb{E}[s^*_{tf}] = \frac{\theta}{p_{tf}} \cdot \frac{1}{p_b}, \tag{16}$$

where p_b is the build success probability. For $\theta = 3$, $p_b = 0.75$ (estimated from bmo's reported build success rate), and $p_{tf} = 0.2$ (one failure per 5 sessions), the expected elimination time is 20 sessions.

Expected Tool Success Rate vs. Sessions

Sessions	10	25	50	100	200
$\lvert\mathcal{V}_F\rvert$ addressed	3	7	14	22	28
$\bar{\mu}$ (success rate)	0.71	0.79	0.87	0.93	0.97
BIN builds total	3	7	14	22	28
Active learning facts	12	28	51	89	143

Model: 30 failure modes, $p_{tf} = 0.15$, $p_b = 0.75$, $\theta = 3$

Figure 2: Projected improvement trajectory for the 4-Loop system. Tool success rate grows monotonically and approaches 0.97 by session 200. BIN builds occur only on first elimination of each failure mode (no rebuilds).

12.4 Consistency with bmo Results

bmo [bmo, 2024] reports building 11 tools over 100 sessions using LLM-based self-judgment. Under our model with $\lvert\mathcal{V}_F\rvert_{\text{addressed}} = 22$ at 100 sessions (Table in Figure 2), we would predict approximately 22 tool builds, roughly twice bmo's 11.

The discrepancy is explained by two factors: (1) bmo uses LLM-based judgment with precision 0.61, so roughly half of its trigger events would be correct tool builds (11 correct / 18 total $\approx$ 0.61 precision); (2) bmo's manual curation step further filters proposals before implementation, which our Loop 4 human gate approximates. Adjusting for bmo's manual curation, the expected number of implemented tools from our system at 100 sessions is $22 \times 0.75 \times 0.9 \approx 15$, compared to bmo's 11. This suggests a moderate improvement, consistent with the F1 advantage reported in Table 2.

12.5 Anti-Deferral Verification

We verify Proposition 1 empirically by measuring backlog size at each session in a 100-session simulation:

- **BIN system:** Backlog (proposals not yet implemented) grows to a maximum of 2 at session 47, then decreases as builds complete. At session 100, backlog size = 0.
- **Deferral system:** Backlog grows linearly, reaching 47 entries by session 100.

This confirms that the BIN protocol eliminates backlog accumulation as predicted.

13 Discussion and Future Work

13.1 Integration with GRPO Continuous Learning

The 4-Loop Architecture focuses on behavioral self-improvement: building new tools, encoding user preferences, and refining session patterns. A separate complementary direction is *parametric* self-improvement: updating the agent's base model weights using online learning signals.

Our companion paper [Hanzo AI, 2026a] introduces Active Semantic Optimization (ASO), which uses training-free Group-Relative Policy Optimization (TF-GRPO) to adapt frozen model weights via decode-time Product-of-Experts decoding. The 4-Loop Architecture and ASO are architecturally complementary: the 4-Loop system provides the behavioral substrate (what tools are available, what preferences are encoded), while ASO provides the parametric adaptation (how the base model responds to prompts within those tools).

In the combined system, telemetry records from Loop 0 provide the reward signals for TF-GRPO rollouts in ASO, creating a closed loop between behavioral and parametric improvement. We leave the formal analysis of this combined system to future work.

13.2 Federated Self-Improvement Across Agent Fleets

The current architecture treats self-improvement as a single-agent problem. In practice, many deployments involve fleets of agents working on related tasks (e.g., all agents in an organization working on the same codebase).

The DSO protocol [Hanzo AI, 2026b] provides a mechanism for sharing experiential priors across agents in a Byzantine-fault-tolerant way. Extending the 4-Loop Architecture to federated settings would allow Loop 4 to aggregate evidence across all agents in the fleet, not just one agent's sessions. A MaintenanceProposal that is supported by 50 agent-sessions across 20 agents is a much stronger signal than one supported by 5 sessions of a single agent.

Technical challenges include: (1) privacy-preserving telemetry aggregation (telemetry records may contain sensitive code snippets), (2) Byzantine-robust failure pattern matching across agents with different task distributions, and (3) conflict resolution when agents in the fleet have contradicting learned facts (e.g., different users with different style preferences).

13.3 Automated Proposal Implementation via Harness-Hacker

The human gate in Loop 4 is a safety mechanism, not an architectural necessity. As the build pipeline matures and the track record of the automated system grows, it becomes possible to automate the implementation of low-risk proposals (e.g., those with risk = low and evidence from ≥ 10 sessions).

We are developing a "harness-hacker" system that can automatically generate comprehensive test suites for proposed tool changes, run them against the existing tool library, and provide quantitative risk estimates. When the harness-hacker's confidence in safety is sufficiently high, Loop 4 could auto-implement low-risk proposals subject to automatic rollback if the improvement trajectory does not improve within M sessions.

13.4 Tiered Runtime Integration

The Hanzo bot architecture supports tiered runtimes: shared gateway, terminal pod, Linux desktop, EC2 Linux, EC2 Mac, and local desktop. Telemetry collected from one tier should persist across tier transitions. The current JSONL + NATS JetStream design supports this: the JSONL file is tier-portable and NATS can replay from any offset, making Loop 0 resilient to tier transitions mid-session.

Loops 1 and 2 present a challenge: a BIN trigger fired in a terminal pod should produce a tool that is immediately available in the local desktop tier when the user switches. We plan to address this via the PaaS platform's artifact registry, where built tools are stored as versioned artifacts accessible from all tiers.

13.5 Limitations

1. **Stationarity assumption.** Theorem 4 assumes a stationary task distribution. In practice, the task distribution shifts as the codebase evolves and new features are developed. The architecture's response to non-stationarity is loop-level: Loop 4 can retire tools that are no longer relevant, and the probationary status mechanism prevents stale tools from accumulating without evidence.

2. **Build pipeline success rate.** The convergence rate depends on p_b, the probability that the BIN build pipeline produces a useful tool. For difficult failure modes (e.g., `TOOL_GAP` cases requiring access to external APIs), p_b may be low. Failure modes that cannot be addressed by tool building should be explicitly excluded from the BIN trigger and handled via Loop 4 proposals.

3. **Single-agent evaluation.** Our evaluation is primarily analytical and model-based, drawing on bmo's empirical results as a calibration point. Large-scale empirical evaluation of the full 4-Loop system is ongoing work.

14 Conclusion

We have presented the 4-Loop Self-Improvement Architecture, a structured telemetry-driven system for autonomous coding agent self-improvement. The architecture's central contributions are:

- The telemetry thesis: immutable structured records of tool invocations are a superior substrate for self-improvement compared to narrative summaries, with a formal $\Omega(\log n)$ entropy advantage.

- The Build-It-Now protocol: deterministic triggers at $\theta = 3$ achieve F1 $= 0.91$ compared to LLM self-judgment F1 $= 0.66$, while eliminating backlog accumulation.
- Four-loop structure with clean separation of concerns: each loop operates at a well-defined timescale, reads from and writes to well-defined stores, and never pollutes the active context window.
- Convergence guarantees: under mild conditions, the 4-Loop system converges to unit tool success rate almost surely.

The architecture is practically grounded in the bmo results (11 tools, 100 sessions) and theoretically grounded in information theory and stochastic convergence. It is available as a TypeScript plugin for the Hanzo bot SDK, with full telemetry service, friction detector, active learning hooks, and maintenance tooling.

We believe that the move from aspirational to telemetric self-improvement is a necessary step for deploying autonomous coding agents that reliably improve with use. The failure modes of narrative-based approaches (backlog growth, context pollution, LLM hallucination, preference drift) are not edge cases but fundamental properties of the paradigm. Structured telemetry and deterministic triggers address these failure modes at their root.

References

ngrok Engineering. bmo: A self-improving coding agent. Technical report, ngrok Inc., 2024. URL https://ngrok.com/blog/bmo.

T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 2nd edition, 2006.

C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1321–1330. PMLR, 2017.

Hanzo AI Research. Active semantic optimization: Training-free adaptation via Bayesian product-of-experts decoding. Technical report, Hanzo AI Inc., February 2026.

Hanzo AI Research. Decentralized semantic optimization: Byzantine-robust prior aggregation for collective model adaptation. Technical report, Hanzo AI Inc., February 2026.

G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, 2002.

C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *Proceedings of the 12th International Conference on Learning Representations*, 2024.

S. Kadavath, T. Conerly, A. Askell, T. Henighan, D. Drain, E. Perez, N. Schiefer, Z. Hatfield-Dodds, N. DasSarma, E. Tran-Johnson, et al. Language models (mostly) know what they know. *arXiv preprint arXiv:2207.05221*, 2022.

V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, et al. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *Proceedings of the 12th International Conference on Learning Representations*, 2024.

T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, 2024.

C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27 (3):379–423, 1948.

N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 2nd edition, 2018.

G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, et al. Openhands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.