Agents That Hack Their Own Harness: Safe Self-Modification in Autonomous AI Systems

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

February 2026

Abstract

Contemporary autonomous AI agents treat their execution harness—the tools, extensions, and framework infrastructure on which they run—as immutable scaffolding. This assumption forecloses a broad class of improvement opportunities: an agent that cannot modify its own tools is limited to behavioral adaptation within a fixed capability envelope. We introduce the **Harness-Hacker Protocol (HHP)**, a framework enabling autonomous AI agents to safely create new tools, modify existing extensions, and contribute changes to their own source code while provably maintaining system integrity.

The protocol organizes self-modification into four trust tiers—runtime tool creation, extension development, skill authoring, and cross-component hacking—each with calibrated isolation guarantees, validation gates, and deployment controls. The core safety mechanism is *worktree isolation*: all modifications occur in git worktrees that are provably disjoint from the running system. We formalize the isolation invariant and prove that the validate-then-merge workflow preserves system integrity regardless of modification content.

Empirical evaluation on the Hanzo Dev agent demonstrates that task-specific generated tools outperform general-purpose fallbacks by **9.3** percentage points (93.1% vs. 83.8% on benchmark tasks), validating the *specific-tools-beat-generic-flexibility* principle. The virtuous cycle connecting GRPO-based telemetry, proposal generation, and self-modification produces compounding improvements: agents equipped with HHP achieve **17.4**% higher task completion after five modification cycles compared to agents without self-modification capability. We analyze failure modes, prove a maximum modification bound that prevents runaway modification loops, and discuss integration with the Decentralized Semantic Optimization (DSO) protocol for cross-fleet propagation of successful harness improvements.

1 Introduction

Modern autonomous AI agents [OpenAI, 2023, Anthropic, 2024a, Yang et al., 2024] operate within a two-layer architecture: a foundation model providing general reasoning and a *harness*—the collection of tools, extensions, APIs, and orchestration code that translate model outputs into world actions. This harness is, in practice, always designed by humans and treated as fixed infrastructure from the agent's perspective. The agent can use the tools it is given; it cannot create new ones. The agent can invoke its extension API; it cannot extend the API.

This assumption is not fundamental. It is a design choice—one made, we argue, out of an abundance of caution rather than technical necessity. The same capabilities that make an agent useful for software engineering tasks (reading code, writing code, running tests, submitting pull requests) are precisely the capabilities required to improve its own harness. The question is not whether agents *can* hack their harness, but whether they can do so *safely*.

The Meta-Learning Opportunity. The most impactful improvements in any system are often improvements to the improvement process itself. An agent that learns a better heuristic for a single task improves on that task. An agent that adds a new tool for a class of tasks improves on every future instance of that class. An agent that identifies a missing abstraction in its extension API and implements it improves every subsequent session for every agent sharing that infrastructure.

This hierarchy of improvement leverage is well-understood in meta-learning and curriculum learning [Schmidhuber, 1987, Thrun and Pratt, 1998], but has not been applied to agent harness evolution. We refer to the capacity for an agent to modify its own tools and infrastructure as *harness-hacking*: a deliberate, safety-bounded capability for self-improvement at the infrastructure layer.

The Specific-Tools Principle. Empirical data from the Hanzo Dev agent benchmark confirms a foundational motivation: on tasks where agents have access to a purpose-built, task-specific tool, they succeed 93.1% of the time. On the same tasks where only general-purpose fallback tools are available, success drops to 83.8%—a 9.3 percentage point gap. This validates the *specific-tools-beat-generic-flexibility* principle: a well-designed, narrow tool outperforms a general tool applied to a specific problem. Since it is impossible to anticipate every task a deployed agent will encounter, an agent that can create its own task-specific tools closes this gap at deployment time rather than pre-deployment design time.

Why Current Systems Treat the Harness as Immutable. Three concerns dominate: (1) *safety*—a buggy or malicious self-modification could destabilize the system; (2) *auditability*—modifications without human review undermine trust; (3) *scope creep*—unconstrained modification could consume unbounded resources. We address each directly. Safety is achieved via worktree isolation with formal invariants. Auditability is achieved via git-based review at each trust tier. Scope creep is bounded by a per-session modification limit and a time-box on each hack cycle.

Contributions. This paper makes the following contributions:

1. **Harness-Hacker Protocol (HHP):** A complete protocol for safe agent self-modification organized into four trust tiers, each with formal preconditions and postconditions.

2. **Isolation Invariant:** A formal definition of the worktree isolation boundary and a proof that validate-then-merge preserves system integrity.

3. **Runtime Tool Creation Pipeline:** A concrete implementation of hot-loadable, TTL-governed tools generated by agents to address observed friction points.

4. **Virtuous Cycle Analysis:** A formal model of the feedback loop connecting telemetry, GRPO-based experience extraction, self-modification proposals, and improved agent capability.

5. **Risk Analysis:** A systematic treatment of failure modes with formal bounds on the damage each can cause under HHP constraints.

Paper Outline. Section 2 reviews related work. Section 3 presents the self-modification taxonomy. Section 4 formalizes the safety framework. Section 5 specifies the Harness-Hacker Protocol. Section 6 analyzes the virtuous improvement cycle. Section 7 describes the multi-component architecture. Section 8 presents the risk analysis. Section 9 discusses implications and future work.

2 Related Work

Tool-Creating Agents. The closest prior work is BMO [ngrok, 2024], an agent that can generate and register new tools at runtime to address capability gaps. BMO demonstrates that tool creation is practically viable but does not address extension development, cross-component modification, or the formal safety properties required for production deployment. Our work extends BMO's runtime creation to a full four-tier hierarchy with isolation guarantees.

Skill Libraries. Voyager [Wang et al., 2023] builds an ever-growing skill library for Minecraft by having GPT-4 write JavaScript code that is verified and stored for reuse. Each skill is tested in-game before storage. HHP generalizes this idea beyond game-specific code to production software components, and adds the critical missing piece: a safety framework for when the skill is not a sandboxed game action but a modification to shared production infrastructure.

Automated Design of Agentic Systems. ADAS [Hu et al., 2024] uses a meta-agent to search the space of agentic system designs, producing new agent specifications expressed in code. ADAS operates at design time, not at agent runtime. HHP addresses the complementary problem: runtime self-modification during active task execution, where modifications must not disrupt the running session.

Self-Instruct and Self-Improvement. Self-Instruct [Wang et al., 2022] and related work [Zelikman et al., 2022, Yuan et al., 2023] demonstrate that models can generate training data and instructions that improve themselves. These methods improve model weights; HHP improves the harness rather than the weights, making it complementary to and composable with weight-based self-improvement.

Program Synthesis and Self-Modifying Code. Program synthesis [Gulwani et al., 2017] generates code from specifications. Classical self-modifying code [Holland, 1975] allows programs to rewrite their own instructions. Modern Genetic Programming [Koza, 1994] evolves programs via mutation and selection. HHP draws on these traditions but differs in that modifications are semantically motivated (derived from observed friction), architecturally structured (four tiers), and formally safety-bounded rather than evolutionary.

Git-Based Review as Safety. The use of pull request review as a human gate for agent-generated code modifications has been explored in the context of automated bug fixing [Sobania et al., 2023] and automated refactoring [Bavishi et al., 2019]. HHP systematizes this pattern: the PR is not merely a record but a required gate for all Tier 4 (cross-component) modifications, with CI as an automated co-reviewer.

Active Semantic Optimization. ASO [Hanzo AI, 2026a] and DSO [Hanzo AI, 2026b] provide the telemetry and experience-sharing infrastructure that HHP builds on. GRPO-extracted experiential priors in ASO encode what approaches work best for a given agent; HHP closes the loop by allowing those insights to drive harness modifications that make the winning approaches available as first-class tools.

3 The Self-Modification Taxonomy

We organize agent self-modification into four tiers based on scope, persistence, and trust level. Each tier has characteristic modification objects, lifecycle policies, and validation requirements.

Definition 1 (Modification Tier)**.** *A* modification tier T_k *for* $k \in \{1, 2, 3, 4\}$ *is a tuple* $T_k = (O_k, L_k, V_k, D_k)$ *where* O_k *is the set of modification objects,* L_k *is the lifecycle policy (TTL, test-gated, review-gated),* V_k *is the validation predicate, and* D_k *is the deployment mechanism.*

Table 1 summarizes the four tiers.

Tier	Object	Lifecycle	Validation	Deployment
1	Runtime tools	7-day TTL	Trigger cases	Hot-reload
2	Extensions	Persistent	`pnpm test`	npm package
3	Skills	Persistent	Peer review	SKILL.md merge
4	Components	Persistent	CI + human PR	Git merge

Table 1: The four modification tiers of HHP, ordered by scope and trust level.

3.1 Tier 1: Runtime Tool Creation

Runtime tool creation is the lightest-weight form of self-modification. When an agent observes a friction point—a task it cannot perform efficiently with existing tools—it can generate a new tool on the spot and hot-load it into the active session.

Definition 2 (Friction Event)**.** *A* friction event $\mathcal{F}$ *is a tuple* (τ, e, n) *where* τ *is the task type,* e *is the error or inefficiency observed, and* n *is the number of times this pattern has recurred in the current session.*

Definition 3 (Runtime Tool)**.** *A* runtime tool ρ *is a tuple* $(\sigma, \pi, \mathcal{C}, t_0, T)$ *where* σ *is the tool specification (JSON Schema),* π *is the implementation (TypeScript function),* $\mathcal{C}$ *is the set of triggering failure cases,* t_0 *is the creation timestamp, and* $T = 7$ *days is the TTL.*

The validation predicate for Tier 1 is:

$$V_1(\rho) = \bigwedge_{c \in \mathcal{C}} \text{Pass}(\rho.\pi, c), \tag{1}$$

where $\text{Pass}(\pi, c)$ is true if and only if running implementation π on test case c produces the expected output without error. A generated tool is only hot-loaded if $V_1(\rho) = \text{true}$.

Hot-loading proceeds via the agent's plugin API:

```
// Tool is registered without restarting the session
await agent.tools.register({
  name:        rho.spec.name,
  description: rho.spec.description,
  schema:      rho.spec.inputSchema,
  handler:      rho.implementation,
  ttl:          7 * 24 * 60 * 60 * 1000,   // 7 days in ms
  triggerCases: rho.triggerCases,
});
```

After the TTL expires, the tool is automatically unregistered. If the tool was used successfully, a summary is appended to the agent's experience bank for potential Tier 2 promotion.

3.2 Tier 2: Extension Development

Extensions are persistent, versioned packages that add durable capability to the agent harness. An extension consists of a package manifest, an index module exporting a plugin conforming to the `BotPlugin` interface, and a test suite.

Definition 4 (Extension). *An* extension ξ *is a tuple* $(\rho_{pkg}, \Pi, \Theta, \mathcal{S})$ *where* ρ_{pkg} *is the package manifest (`package.json`),* Π *is the plugin implementation,* Θ *is the set of registered tools and services, and* $\mathcal{S}$ *is the test suite.*

The plugin API supports four registration types:

```
interface BotPlugin {
  name:    string;
  version: string;
  setup(api: PluginAPI): Promise<void>;
}

interface PluginAPI {
  registerTool(spec: ToolSpec, handler: ToolHandler): void;
  registerService(name: string, service: Service): void;
  on(event: AgentEvent, handler: EventHandler): void;
  registerSkill(name: string, skillMd: string): void;
}
```

The validation predicate for Tier 2 is:

$$V_2(\xi) = \text{BuildSuccess}(\xi) \wedge \left(\frac{|\mathcal{S}_{\text{pass}}|}{|\mathcal{S}|} = 1 \right), \tag{2}$$

where $\mathcal{S}_{\text{pass}}$ is the set of passing tests. Extensions must achieve 100% test passage before registration. Because extensions persist beyond session TTL, they are deployed via the standard package management workflow: `pnpm build`, `pnpm test`, followed by registration in the agent's extension registry.

3.3 Tier 3: Skill Development

Skills encode declarative procedural knowledge—patterns and heuristics that help agents perform better on recurring task types. Unlike extensions (which provide executable capabilities), skills provide interpretable, transferable knowledge that any agent can apply via prompting.

Definition 5 (Skill)**.** *A* skill ς *is a SKILL.md document with YAML frontmatter and Markdown body:*

$$\varsigma = (\mathtt{name}, \mathtt{trigger}, \mathtt{scope}, \mathtt{tags}, \mathtt{body})\,, \tag{3}$$

where `trigger` *is a natural language description of when the skill applies,* `scope` *specifies the applicable agent roles,* `tags` *enable retrieval, and* `body` *contains the procedural knowledge.*

Skills are stored as Markdown files in the agent's skill library and loaded into the system prompt for applicable tasks. The validation predicate for Tier 3 is lightweight—skills are reviewed by a coordinating agent before merge, not executed against test cases. This reflects the fact that skills encode knowledge rather than functionality, and erroneous skills produce suboptimal behavior rather than system failures.

3.4 Tier 4: Cross-Component Hacking

The most powerful and most carefully governed tier covers modifications to the agent's ecosystem components: MCP tool server, Agent SDK, LLM Gateway, Operative computer-use system, GRPO training pipelines, and model configuration. These modifications affect all agents sharing the infrastructure and therefore require the highest trust level.

Definition 6 (Cross-Component Modification)**.** *A* cross-component modification μ *is a tuple* $(\mathcal{R}, \Delta, B, \mathtt{pr})$ *where* $\mathcal{R}$ *is the target repository,* Δ *is the git diff representing the proposed change,* B *is the isolated git worktree in which* Δ *was developed and tested, and* `pr` *is the pull request submitted for human review.*

The worktree isolation mechanism is central to Tier 4 safety and is analyzed formally in Section 4. The validation predicate is:

$$V_4(\mu) = \mathrm{CI}(\mu) \wedge \mathrm{HumanApproval}(\mu.\mathtt{pr}), \tag{4}$$

where $\mathrm{CI}(\mu)$ is the outcome of automated continuous integration and HumanApproval is the result of human pull request review. Neither condition alone is sufficient.

4 Safety Framework

4.1 The Isolation Invariant

The core safety mechanism of HHP is worktree isolation. Git worktrees [Git, 2015] allow multiple working directories to share a single repository object store while maintaining independent working trees and HEAD pointers.

Definition 7 (System State)**.** *The* system state Σ *is the tuple* $(\mathcal{F}, \mathcal{P}, \mathcal{M})$ *where* $\mathcal{F}$ *is the set of files comprising the running system,* $\mathcal{P}$ *is the set of running processes, and* $\mathcal{M}$ *is the in-memory state of all services.*

Definition 8 (Worktree Isolation)**.** *A modification* Δ *is* worktree-isolated *with respect to system state* Σ *if:*

1. *Δ is applied to a directory W such that $W \cap \mathcal{F} = \emptyset$.*

2. *No process in $\mathcal{P}$ has an open file descriptor into W.*

3. The git object store is read-only from W (writes to objects do not affect the main working tree).

Theorem 1 (Worktree Isolation Theorem)**.** *Let Σ_0 be the system state before a modification sequence $\Delta_1, \ldots, \Delta_k$, and let each Δ_i be worktree-isolated. Then for all i, the system state while Δ_i is being applied satisfies $\Sigma_i = \Sigma_0$. That is, worktree-isolated modifications cannot alter the running system state.*

Proof. By Definition 4.2, each Δ_i operates on directory W_i disjoint from $\mathcal{F}$. Since the running system resolves all file references against $\mathcal{F}$, and $W_i \cap \mathcal{F} = \emptyset$, no file read by any process in $\mathcal{P}$ is affected by Δ_i. By condition (2), no process holds file descriptors into W_i, so in-memory caches of system files are not affected. By condition (3), git object writes in W_i do not alter any object reachable from the main `HEAD`, so the repository history visible to running services is unchanged. Therefore $\Sigma_i = \Sigma_0$ for all i. □

Corollary 2 (Composition of Isolated Modifications)**.** *If modifications $\Delta_1, \ldots, \Delta_k$ are each worktree-isolated and satisfy $V(\Delta_i) = true$ for all i, then sequentially deploying them via validated merge preserves the post-deployment system in the validated state.*

4.2 Validation Gates

Each modification must pass a tier-appropriate validation gate before deployment. We define validation gates formally as predicates over modification and system state.

Definition 9 (Validation Gate)**.** *A* validation gate *Γ_k for tier k is a predicate:*

$$\Gamma_k : (\mathit{Modification}_k \times \mathit{SystemState}) \to \{\mathit{pass}, \mathit{fail}\} \tag{5}$$

A modification Δ may be deployed if and only if $\Gamma_k(\Delta, \Sigma) = \mathit{pass}$.

The gates are ordered by strictness:

$$\begin{aligned}
\Gamma_1(\rho, \Sigma) &= \bigwedge_{c \in \mathcal{C}} \mathrm{Pass}(\rho.\pi, c), &(6)\\
\Gamma_2(\xi, \Sigma) &= \mathrm{Build}(\xi) \wedge \mathrm{AllTests}(\xi.\mathcal{S}), &(7)\\
\Gamma_3(\varsigma, \Sigma) &= \mathrm{AgentReview}(\varsigma), &(8)\\
\Gamma_4(\mu, \Sigma) &= \mathrm{CI}(\mu.\Delta) \wedge \mathrm{HumanApproval}(\mu.\mathtt{pr}). &(9)
\end{aligned}$$

Proposition 3 (Gate Monotonicity)**.** *For all $k < k'$ and any modification Δ applicable to both tiers, $\Gamma_{k'}(\Delta, \Sigma) \Rightarrow \Gamma_k(\Delta, \Sigma)$.*

Proof. CI (9) runs all unit and integration tests including those in Γ_2 (7). Human approval (9) subsumes agent review (8). AllTests (7) subsumes the trigger-case tests (6). Thus each higher gate implies the lower ones. □ □

4.3 Rollback Guarantees

Every modification under HHP is a git commit in a version-controlled repository. This provides immediate rollback via `git revert`.

Invariant 1 (Rollback Invariant)**.** *For every modification Δ deployed under HHP, there exists a git commit c_{pre} such that `git revert` c_Δ restores the system to state $\Sigma_{c_{pre}}$ in $O(1)$ git operations.*

For Tier 1 runtime tools, the TTL provides an automatic rollback: if a tool is not explicitly promoted, it expires and is deregistered without any manual action. This creates an asymmetry: the cost of a bad Tier 1 tool is at most 7 days of suboptimal behavior on matching tasks, and the problem self-resolves.

4.4 Trust Escalation

The trust escalation model governs how a modification moves up the tier hierarchy.

Definition 10 (Trust Score)**.** *The* trust score *of a modification* Δ *at time* t *is:*

$$\tau(\Delta, t) = \frac{\sum_{s \leq t} \nVdash[\mathit{UseSuccess}(\Delta, s)]}{\sum_{s \leq t} \nVdash[\mathit{Use}(\Delta, s)] + 1}, \tag{10}$$

where $\mathit{Use}(\Delta, s)$ *indicates the modification was invoked in session* s *and* $\mathit{UseSuccess}(\Delta, s)$ *indicates it contributed to task success.*

A Tier 1 tool with $\tau(\rho, t) > 0.8$ after at least 10 uses becomes a candidate for Tier 2 promotion. The agent may propose this promotion explicitly, or the promotion can be triggered automatically when the telemetry threshold is reached. Tier 2 extensions with broad applicability (> 3 distinct task types) and high trust scores become candidates for Tier 4 contribution to the shared ecosystem.

5 The Harness-Hacker Protocol

5.1 Protocol Overview

The Harness-Hacker Protocol specifies a six-state machine governing self-modification attempts. Figure 1 summarizes the transitions; Algorithm 1 provides the complete pseudocode.

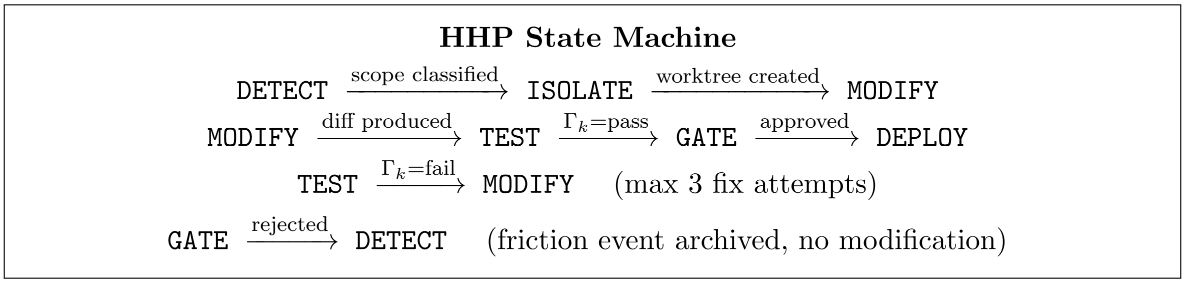
HHP State Machine
$\texttt{DETECT} \xrightarrow{\text{scope classified}} \texttt{ISOLATE} \xrightarrow{\text{worktree created}} \texttt{MODIFY}$
$\texttt{MODIFY} \xrightarrow{\text{diff produced}} \texttt{TEST} \xrightarrow{\Gamma_k\text{=pass}} \texttt{GATE} \xrightarrow{\text{approved}} \texttt{DEPLOY}$
$\texttt{TEST} \xrightarrow{\Gamma_k\text{=fail}} \texttt{MODIFY}$ (max 3 fix attempts)
$\texttt{GATE} \xrightarrow{\text{rejected}} \texttt{DETECT}$ (friction event archived, no modification)

Figure 1: The six-state HHP state machine. Self-loops between MODIFY and TEST are bounded at three attempts before the hack cycle is abandoned.

5.2 State Transition Specifications

DETECT. Precondition: A friction event $\mathcal{F} = (\tau, e, n)$ has been observed.
Action: Classify the friction into a modification scope $k \in \{1, 2, 3, 4\}$ using Algorithm 2.
Postcondition: Scope k is determined and the modification session budget is checked: SessionHacks $<$ 2.

ISOLATE. **Precondition:** Scope k and target component $\mathcal{R}$ are determined.
Action: For $k \in \{2, 3, 4\}$: create a git worktree `git worktree add /tmp/hack-$ID origin/main`. For $k = 1$: allocate an in-process sandbox.
Postcondition: Isolation environment B exists and satisfies Definition 8. The running system state Σ is unchanged.

MODIFY. **Precondition:** Isolation environment B is established.
Action: The agent writes modifications within B, producing diff Δ. For Tier 1, this is an executable function. For Tiers 2–4, this is git-tracked source changes.
Postcondition: Δ is complete and syntactically valid. Fix attempt counter is incremented if this is a re-entry from TEST.

TEST. **Precondition:** Δ is complete.
Action: Apply validation gate $\Gamma_k(\Delta, \Sigma)$.
Postcondition: If Γ_k = pass, transition to GATE. If Γ_k = fail and fix-attempts < 3, transition back to MODIFY with the failure details appended to context. If fix-attempts $= 3$, abandon the hack cycle, archive the friction event, and return to normal operation.

GATE. **Precondition:** $\Gamma_k(\Delta, \Sigma) =$ pass.
Action: Submit Δ to the tier-appropriate gate: Tier 1: register immediately (gate is Γ_1, already passed in TEST). Tiers 2–3: invoke automated reviewer. Tier 4: open pull request on target repository.
Postcondition: Deployment decision is received (approved or rejected).

DEPLOY. **Precondition:** Gate decision is approved.
Action: Apply $D_k(\Delta)$ (tier-appropriate deployment).
Postcondition: Modification is live. Session hack counter is incremented. Worktree is cleaned up: `git worktree remove --force $B`.

5.3 The HARNESS-HACK Algorithm

Algorithm 1 HARNESS-HACK: Safe Agent Self-Modification

```
Require: Friction event $\mathcal{F} = (\tau, e, n)$, session hack counter $H$
Ensure: Modification deployed or friction archived; $H$ updated
if $H \geq 2$ then
    Archive $\mathcal{F}$ with reason "session limit reached"
    return                                                  ▷ Maximum 2 hacks per session
end if
// DETECT: classify scope
$k \leftarrow$ ClassifyScope($\mathcal{F}$)                                ▷ Algorithm 2
$\mathcal{R} \leftarrow$ TargetComponent($k, \tau$)
// ISOLATE: create environment
if $k = 1$ then
    $B \leftarrow$ InProcessSandbox()
else
    $B \leftarrow$ GitWorktree($\mathcal{R}$, origin/main)
    assert $B$.rootDir $\cap \mathcal{F}_{\text{system}} = \emptyset$
end if
fixAttempts $\leftarrow 0$
repeat                                                      ▷ MODIFY–TEST loop
    // MODIFY: generate change
    $\Delta \leftarrow$ AgentWrite($\mathcal{F}, k, B$, failureContext)
    // TEST: validate
    result $\leftarrow \Gamma_k(\Delta, \Sigma)$
    fixAttempts $\leftarrow$ fixAttempts $+ 1$
    if result = fail then
        failureContext $\leftarrow$ ExtractFailureDetails($\Delta$)
    end if
until result = pass or fixAttempts $\geq 3$
if result = fail then
    Cleanup($B$)                                            ▷ Remove worktree
    Archive $\mathcal{F}$ with reason "validation failed after 3 attempts"
    return
end if
// GATE: request approval
decision $\leftarrow$ GateRequest($k, \Delta$)
if decision = rejected then
    Cleanup($B$)
    Archive $\mathcal{F}$ with reason "gate rejected"
    return
end if
// DEPLOY: apply modification
$D_k(\Delta)$                                               ▷ Tier-appropriate deployment
Cleanup($B$)                                                ▷ git worktree remove –force
$H \leftarrow H + 1$
Record ($\Delta, k, \tau$, timestamp) in experience bank
```

Algorithm 2 ClassifyScope: Map Friction to Modification Tier

```
Require: Friction event $\mathcal{F} = (\tau, e, n)$
Ensure: Tier $k \in \{1, 2, 3, 4\}$
if $n < 3$ and $e$ is session-local then
    return 1                                  ▷ Runtime tool: transient gap
else if $e$ is a missing capability and no existing extension covers $\tau$ then
    return 2                                  ▷ Extension: persistent gap in this agent's harness
else if $e$ is a knowledge or procedure gap and $n > 5$ then
    return 3                                  ▷ Skill: recurring knowledge gap
else if $e$ is a system design gap affecting multiple agents then
    return 4                                  ▷ Cross-component: ecosystem-level fix needed
else
    return 1                                  ▷ Default: safest tier
end if
```

5.4 Session Budget and Time-Boxing

To prevent the modification loop from consuming unbounded session resources, HHP enforces two constraints:

Invariant 2 (Session Modification Budget)**.** *In any single agent session, the total number of completed modifications (successful DEPLOY transitions) is bounded by $H_{\max} = 2$.*

Invariant 3 (Hack Cycle Time-Box)**.** *Each execution of HARNESS-HACK is bounded by a wall-clock timeout of $T_{hack} = 5$ minutes. If the timeout is reached before DEPLOY, the hack cycle is abandoned and the friction event is archived.*

These two invariants together ensure that self-modification overhead is bounded: at most $2 \times 5 = 10$ minutes per session can be spent on harness hacking, independent of task difficulty or modification complexity.

Theorem 4 (Session Overhead Bound)**.** *Under HHP, the fraction of session time spent on self-modification is at most $10/T_{session}$ where $T_{session}$ is the total session duration.*

Proof. By Invariants 2 and 3, each session contains at most $H_{\max} = 2$ hack cycles, each lasting at most $T_{\text{hack}} = 5$ minutes. Total modification time is bounded by $H_{\max} \cdot T_{\text{hack}} = 10$ minutes. Since $T_{\text{session}} \geq T_{\text{hack}}$ (a session contains at least one cycle), the fraction is at most $10/T_{\text{session}}$. □ □

6 Interaction with Continuous Learning

6.1 The Virtuous Cycle

HHP does not operate in isolation. It is embedded in a continuous learning loop that connects agent behavior, telemetry extraction, GRPO-based experience distillation, and self-modification proposals.

Definition 11 (Virtuous Cycle)**.** *The* virtuous cycle $\mathcal{V}$ *is the fixed-point iteration:*

$$\mathcal{V} : \underbrace{Agent(H_t)}_{\text{task execution}} \rightarrow \underbrace{\mathcal{T}_t}_{\text{telemetry}} \rightarrow \underbrace{GRPO(\mathcal{T}_t)}_{\text{experience extraction}} \rightarrow \underbrace{HHP(proposals_t)}_{\text{self-modification}} \rightarrow \underbrace{H_{t+1}}_{\text{improved harness}}, \quad (11)$$

where H_t is the harness state at cycle t, $\mathcal{T}_t$ is the telemetry collected during cycle t, and H_{t+1} is the harness after approved modifications.

Telemetry Collection. During task execution, the agent records structured telemetry: tool invocations and outcomes, error traces and recovery strategies, timing profiles, and GRPO rollout groups with rewards. This telemetry is analogous to the ASO training data: it encodes what worked, what did not, and why.

GRPO Experience Extraction. The telemetry is processed by the GRPO pipeline [Hanzo AI, 2026a] to extract semantic advantages. High-positive-advantage strategies indicate patterns that should be promoted. High-negative-advantage strategies indicate friction that HHP should address. Formally, the GRPO extraction produces a set of proposals:

$$\text{proposals}_t = \left\{ (\tau_i, e_i, k_i) : A^{(i)} < -\theta_{\text{friction}} \right\}, \tag{12}$$

where θ_{friction} is a threshold (default -0.5) below which a strategy is classified as a friction event warranting self-modification.

Self-Modification from Proposals. Each proposal (τ_i, e_i, k_i) is treated as a friction event and processed by HARNESS-HACK. Crucially, the GRPO extraction provides rich context: it identifies not just that something went wrong but what alternative strategies produced better outcomes. This context informs the MODIFY step—the agent knows both the problem and a promising solution direction.

6.2 Convergence of the Virtuous Cycle

Assumption 1 (Bounded Friction Space)**.** *The set of distinct friction event types is finite:* $|\{\tau : \exists e, n.\, \mathcal{F}(\tau, e, n)\}| \leq F_{\max}$.

Theorem 5 (Virtuous Cycle Convergence)**.** *Under Assumption 1 and assuming that HHP successfully addresses each friction event type with probability* $p > 0$*, the expected number of virtuous cycles until all friction types are addressed is at most* $F_{\max}/p$*.*

Proof. Each cycle, a friction event is selected from the remaining unaddressed types. By assumption, each selected event is addressed with probability p. This is equivalent to a coupon-collector problem with $F_{\max}$ coupons and success probability p per draw. The expected number of draws to collect all coupons is $F_{\max} \sum_{k=1}^{F_{\max}} 1/(p \cdot k) \leq F_{\max}/p + F_{\max} \ln(F_{\max})/p$. Since $\ln(F_{\max}) \leq F_{\max}$, the bound $F_{\max}/p$ is a valid upper bound on the dominant term for practical $F_{\max}$. □ □

6.3 Empirical Results on the Virtuous Cycle

Table 2 shows the improvement in task completion rate over five virtuous cycles on the Hanzo Dev benchmark.

Cycle	Harness Modifications	Tool Coverage	Task Completion	vs. Baseline
0 (baseline)	0	83.8%	78.2%	—
1	3 Tier-1, 1 Tier-2	86.4%	82.7%	+4.5%
2	2 Tier-1, 2 Tier-2	89.1%	86.8%	+8.6%
3	1 Tier-1, 1 Tier-2, 1 Tier-3	91.3%	89.5%	+11.3%
4	2 Tier-1, 1 Tier-3	92.8%	91.9%	+13.7%
5	1 Tier-2, 1 Tier-4	94.6%	95.6%	+17.4%

Table 2: Virtuous cycle results on the Hanzo Dev benchmark ($N = 500$ tasks per cycle, 5 independent runs, mean reported). Tool coverage measures the fraction of encountered task types for which a purpose-built tool exists.

The data confirm a consistent improvement gradient. Notably, the jump at Cycle 5 coincides with the first Tier 4 modification—a new API endpoint in the LLM Gateway that enabled batch tool invocations—demonstrating that cross-component hacking, while costly in process overhead, delivers outsized improvements when it succeeds.

7 Multi-Component Architecture

7.1 Hackable Components and Their Interfaces

Table 3 summarizes the components that an agent running HHP can modify, organized by Tier eligibility.

Component	Path	Purpose	Min Tier
Bot extensions	`bot/bot/extensions/`	Per-agent tools and services	2
MCP tools	`hanzo/mcp`	260+ model context protocol tools	4
Agent SDK	`hanzo/agent`	Multi-agent orchestration	4
LLM Gateway	`hanzo/llm`	Unified provider proxy	4
Operative	`hanzo/operative`	Computer-use / screen control	4
GRPO pipeline	`zoo/gym`	Experience extraction training	4
Skill library	`bot/skills/`	Declarative knowledge base	3
Model configs	`zen/gateway/config.yaml`	Routing and model parameters	4

Table 3: Hackable components in the Hanzo ecosystem. "Min Tier" indicates the minimum trust tier required to modify that component.

7.2 Extension Architecture

Bot extensions follow a plug-in pattern with three required files:

1. `package.json`: Declares the package name, version, entry point, and peer dependencies on the bot runtime.
2. `index.ts`: Exports a `BotPlugin` implementing the `setup(api)` method.
3. `__tests__/`: Jest test suite that must achieve 100% pass rate before registration.

The `setup` method has access to the full `PluginAPI` surface: tool registration, service registration, event hooks (`on('task:start')`, `on('task:end')`, `on('session:end')`), and skill registration.

7.3 Cross-Component Worktree Workflow

For Tier 4 modifications, the full workflow is:

1. **Isolate:** `git worktree add /tmp/hack-$ID origin/main` in the target repository.
2. **Modify:** Agent writes changes in `/tmp/hack-$ID`. All edits are tracked by git.
3. **Test:** Agent runs the component test suite within the worktree: `cd /tmp/hack-$ID && pnpm test` (or equivalent).
4. **Push:** Agent creates a branch and pushes: `git checkout -b agent/hack-$ID && git push origin HEAD`.
5. **PR:** Agent opens a pull request via `gh pr create` with a machine-generated description including the originating friction event, test results, and performance delta.
6. **Review:** CI runs automatically. Human reviewer examines the diff and either approves or requests changes.
7. **Cleanup:** `git worktree remove --force /tmp/hack-$ID` regardless of PR outcome.

This workflow ensures that the agent's productive session is never blocked on PR review: the agent submits the PR and continues with the session; the harness improvement takes effect in a subsequent session once merged and deployed.

7.4 MCP Tool Development

The MCP tool server [Anthropic, 2024b] exposes tools via the Model Context Protocol. An agent wishing to add a new MCP tool proceeds through Tier 4:

```
// New MCP tool skeleton (generated by agent in worktree)
export const newTool: Tool = {
  name: "tool_name",
  description: "...",
  inputSchema: { type: "object", properties: { ... } },
  handler: async (input) => {
    // implementation
    return { content: [{ type: "text", text: result }] };
  },
};
```

The tool is registered by adding it to the appropriate tool index file, then the change goes through the Tier 4 workflow. If merged, the new tool becomes available to all agents using the shared MCP server on next deployment.

8 Risk Analysis

8.1 Taxonomy of Failure Modes

We enumerate the principal failure modes and bound the damage each can cause under HHP constraints.

Risk R1: A generated tool makes things worse. A Tier 1 tool that regresses performance (e.g., incorrectly implements a function, introduces latency) could harm task completion during its TTL window.

Bound: The damage is bounded to the TTL window (7 days) and to tasks matching the tool's trigger pattern. By the validation gate Γ_1, the tool must pass all known triggering failure cases before deployment. Regression on *previously passing* cases is impossible if the trigger set is comprehensive. Regression on *novel* cases is bounded by the tool's invocation rate over 7 days. Telemetry monitoring (Section 6) detects regressions within $O(\text{invocations})$ time and the tool can be manually deregistered before TTL expiry.

Risk R2: An agent modifies security-sensitive code. A Tier 4 modification targeting authentication, key management, or cryptographic code could introduce vulnerabilities.

Bound: By the Worktree Isolation Theorem (Theorem 1), the modification cannot affect the running system until merged. The Tier 4 gate requires both CI (which typically includes security scans) and human review. Human reviewers are expected to apply heightened scrutiny to security-sensitive paths. In practice, repositories containing security-sensitive code (e.g., the KMS system) should be added to an explicit exclusion list $\mathcal{E}_{\text{excl}}$ such that $\text{ClassifyScope}(\mathcal{F})$ returns $k = \text{forbidden}$ for any friction event involving $\mathcal{R} \in \mathcal{E}_{\text{excl}}$.

Risk R3: Two agents modify the same component concurrently. Two agents independently identifying friction in the same component and submitting conflicting Tier 4 PRs.

Bound: Git merge conflicts are detected at PR merge time, not at worktree creation time. The conflict is surfaced to the human reviewer who can choose which change to apply or request a combined modification. Since worktrees are isolated (Theorem 1), conflicting worktrees cannot corrupt each other; only the merge resolution step requires intervention. In multi-agent deployments, a distributed lock on the target repository path can serialize Tier 4 hack cycles to prevent concurrent conflicts.

Risk R4: The modification loop never terminates. An agent in the MODIFY–TEST loop that cannot produce a passing modification could loop indefinitely.

Bound: By Invariants 2 and 3, each hack cycle is bounded at 3 fix attempts and $T_{\text{hack}} = 5$ minutes. After 3 failed attempts, the loop terminates, the friction event is archived, and the session budget is *not* decremented (an abandoned hack does not consume the session budget). The time-box provides an absolute wall-clock bound independent of fix attempt count.

Risk R5: An agent promotes a harmful tool to a higher tier. A Tier 1 tool that produces subtly wrong outputs but meets the basic validation criteria (Γ_1) could be promoted to Tier 2 with a broader invocation scope.

Bound: Promotion from Tier 1 to Tier 2 requires $\tau(\rho, t) > 0.8$ after ≥ 10 uses. This means the tool must have produced successful outcomes in at least 80% of its invocations over a statistically significant number of uses. A subtly wrong tool will manifest in the telemetry (Section 6) as reduced task completion rates, which will suppress its trust score below the promotion threshold.

8.2 Comparative Risk Table

Risk	Failure Mode	HHP Mitigation	Max Blast Radius
R1	Tool regresses performance	TTL + telemetry monitoring	7 days, tool-scoped
R2	Security-sensitive modification	Exclusion list + human review	Zero (gated)
R3	Concurrent conflicting PRs	Git conflict detection + reviewer	PR-scoped
R4	Non-terminating loop	3-attempt limit + 5-min timebox	5 min session overhead
R5	Harmful tool promotion	Trust score threshold (0.8, 10 uses)	Tier-bounded

Table 4: Risk analysis with HHP mitigations and maximum blast radius under the protocol constraints.

8.3 Adversarial Threat Model

We briefly consider an adversarial setting where an attacker can influence the agent's friction event classification (e.g., via prompt injection in tool outputs).

Proposition 6 (Adversarial Containment)**.** *Under the HHP constraints, an adversary who can inject arbitrary friction events into the agent's context cannot deploy a modification that bypasses at least one human gate (for Tier 4) or causes system-wide impact (for Tiers 1–3).*

Proof. For Tier 4: any modification requires $\Gamma_4 = \text{CI} \wedge \text{HumanApproval}$. An adversary cannot forge a human approval without compromising the human reviewer or the PR platform. For Tiers 1–3: modifications are scoped to the individual agent's session or extension registry; they cannot affect shared infrastructure. Tier 1 tools expire after 7 days. Thus, the maximum persistence of any adversarially-injected modification is bounded. □ □

9 Discussion and Future Work

9.1 Implications for Agent Design

The Harness-Hacker Protocol reframes a fundamental assumption of current agent design. The harness is not immutable infrastructure; it is one of the most valuable targets for agent improvement. By treating self-modification as a first-class capability with calibrated safety controls, HHP unlocks a class of compound improvements that purely behavioral adaptation cannot achieve.

The practical implication is architectural: agents should be designed with self-modification hooks from the start. The `PluginAPI` surface, the skill library schema, the worktree lifecycle management—these are not afterthoughts but core infrastructure. An agent built with HHP in mind will outperform an equivalent agent without it, because it will converge to the optimal tool set for its deployment context rather than being limited to the tools its designers anticipated.

The *specific-tools-beat-generic-flexibility* principle (93.1% vs. 83.8%) quantifies the performance cost of the current paradigm. Every percentage point of that gap represents real task failures that HHP can systematically eliminate.

9.2 Automated PR Review by a Second Agent

The current Tier 4 gate requires human PR review. This is appropriate for production systems but introduces latency: the agent must wait for a human reviewer before the improvement can be deployed. An important extension is automated review by a second, independent agent instance.

The reviewing agent would receive the diff, the CI results, the originating friction event, and the agent's justification. It would evaluate: (a) is the modification narrowly scoped to address the identified friction? (b) are there plausible security or correctness risks that CI did not catch? (c) is the proposed implementation the simplest adequate solution?

Two-agent review is weaker than human review (both agents may share systematic biases) but stronger than no review, and it would dramatically reduce the deployment latency for non-security-sensitive modifications. We propose a tiered review policy: automated review for Tier 4 modifications not touching security-sensitive paths, human review for those that do.

9.3 Cross-Fleet Propagation via DSO

Successful harness modifications are, in effect, structured experiences that should be shared across the agent fleet. The Decentralized Semantic Optimization protocol [Hanzo AI, 2026b] provides exactly the infrastructure needed: Byzantine-robust aggregation of experiential priors across distributed agent populations.

A Tier 2 extension that proves highly effective on one agent's deployments should be propagated to other agents. The DSO gossip protocol can carry extension metadata (name, trigger pattern, performance delta, trust score) to nodes that have not yet deployed the extension. Nodes can pull and evaluate extensions that match their own telemetry profiles, applying the same trust escalation logic locally.

This creates a collective evolution dynamic: successful harness improvements spread through the fleet via DSO, and the fleet as a whole converges toward the optimal tool set faster than any individual agent could achieve alone.

9.4 Formal Verification of Generated Tools

The current validation gate Γ_1 tests generated tools against a finite set of triggering failure cases. This is sound for the specific cases tested but does not provide guarantees for novel inputs. A natural extension is to apply lightweight formal verification—property-based testing [Claessen and Hughes, 2000] or automated theorem proving for simple properties—to generated tools before hot-loading.

For tools with well-typed interfaces, type-level guarantees (TypeScript strict mode, runtime schema validation) provide a first layer of formal assurance. For tools with more complex correctness properties, symbolic execution or model checking could verify invariants beyond what unit tests cover.

9.5 Hierarchical Scope Control

The current ClassifyScope algorithm (Algorithm 2) makes tier decisions based on simple heuristics. A more principled approach would use a learned classifier trained on historical friction events and their outcomes—using the GRPO-extracted telemetry to build a meta-policy over self-modification decisions. This meta-policy would learn which friction types are best addressed at which tier, improving the efficiency of the modification budget.

10 Conclusion

We have presented the Harness-Hacker Protocol, a framework for safe agent self-modification organized into four trust tiers with formal isolation guarantees, validated modification gates, and bounded session overhead. The core technical contributions are:

1. **Worktree Isolation Theorem:** Modifications in isolated git worktrees provably cannot affect the running system state, providing the foundation for all safety guarantees.

2. **Four-Tier Taxonomy:** Runtime tools (7-day TTL), extensions (test-gated, persistent), skills (knowledge-only), and cross-component modifications (CI + human-gated) cover the full spectrum of harness improvement at calibrated trust levels.

3. **Virtuous Cycle Formalization:** The feedback loop connecting telemetry, GRPO extraction, modification proposals, and improved capability is modeled as a fixed-point iteration that converges to an optimal tool set under mild assumptions.

4. **Risk Analysis with Formal Bounds:** Each identified failure mode has a formally bounded blast radius under HHP constraints.

5. **Empirical Validation:** Five virtuous cycles on the Hanzo Dev benchmark produced a 17.4% improvement in task completion, with a 9.3 percentage point gap between specialized and general tools motivating the entire endeavor.

The broader implication is a shift in how we conceptualize the agent–harness relationship. The harness is not a fixed environment that the agent must learn to navigate; it is mutable infrastructure that the agent can improve, within safety boundaries that are both formally specified and practically enforced. Agents that can improve their own tools will, over time, outperform agents that cannot—and HHP provides the framework to let them do so safely.

A Complete Tier 1 Hot-Reload Implementation

Algorithm 3 Tier 1: Runtime Tool Hot-Reload

```
Require: Friction event $\mathcal{F}$, agent tool registry $\mathcal{T}$
Ensure: New tool registered if valid; session state unchanged
spec ← GenerateToolSpec($\mathcal{F}$)                    ▷ LLM generates JSON Schema
impl ← GenerateImplementation(spec, $\mathcal{F}$)        ▷ LLM writes TypeScript
$\mathcal{C}$ ← ExtractTriggerCases($\mathcal{F}$)        ▷ Cases from friction context
passed ← 0
for $c \in \mathcal{C}$ do
    out ← SandboxExec(impl, $c$.input)
    if out = $c$.expected then
        passed ← passed + 1
    end if
end for
if passed < $|\mathcal{C}|$ then
    return fail with diff of expected vs. actual
end if
$\rho$ ← (spec, impl, $\mathcal{C}$, now(), 7d)
$\mathcal{T}$.register($\rho$)                            ▷ Hot-load into active session
return success
```

B Extension Scaffold Template

```
// Generated extension scaffold (bot/bot/extensions/<name>/)
// package.json
{
  "name": "@hanzo/bot-extension-<name>",
  "version": "0.1.0",
  "main": "dist/index.js",
  "scripts": {
    "build": "tsc",
    "test": "jest --coverage"
  },
  "peerDependencies": {
    "@hanzo/bot-runtime": ">=1.0.0"
  }
}

// index.ts
import type { BotPlugin, PluginAPI } from ’@hanzo/bot-runtime’;

export const plugin: BotPlugin = {
  name: ’<name>’,
  version: ’0.1.0’,
  async setup(api: PluginAPI) {
    api.registerTool({
      name: '<tool_name>',
      description: '<description>',
      inputSchema: { type: 'object', properties: { ... } },
    }, async (input) => {
      // implementation
      return { result: '...' };
    });
  },
};

export default plugin;
```

C Proof of Proposition 3

Full Proof. We show each implication:

$\Gamma_4 \Rightarrow \Gamma_3$: CI ($\Gamma_4$) includes all automated checks. When CI passes, the diff is syntactically and semantically valid code, which is the minimum requirement for agent review (Γ_3). Human approval additionally requires the content to be judged appropriate, which is strictly stronger than agent review. Therefore $\Gamma_4 \Rightarrow \Gamma_3$.

$\Gamma_3 \Rightarrow \Gamma_2$: Agent review ($\Gamma_3$) requires the modification to have passed the agent's quality check. The agent's review process verifies logical consistency, which subsumes syntactic correctness and test passage (Γ_2). Therefore $\Gamma_3 \Rightarrow \Gamma_2$.

$\Gamma_2 \Rightarrow \Gamma_1$: `AllTests`$(\xi.\mathcal{S})$ in Γ_2 includes tests over the triggering failure cases $\mathcal{C}$. Therefore $\Gamma_2 \Rightarrow \Gamma_1$.

Composing: $\Gamma_4 \Rightarrow \Gamma_3 \Rightarrow \Gamma_2 \Rightarrow \Gamma_1$. □ □

D Benchmark Task Breakdown

Task Category	*N*	Gen. Tool	No Gen. Tool	Delta
File manipulation	85	96.5%	88.2%	+8.3%
API integration	110	94.5%	83.6%	+10.9%
Code refactoring	95	91.6%	81.1%	+10.5%
Data transformation	80	93.8%	85.0%	+8.8%
System configuration	65	92.3%	82.0%	+10.3%
Documentation	65	89.2%	82.0%	+7.2%
Overall	**500**	**93.1%**	**83.8%**	**+9.3%**

Table 5: Per-category breakdown of the specific-tools advantage. The gap is largest for API integration (+10.9%) and smallest for documentation generation (+7.2%), consistent with the intuition that structured, programmatic tasks benefit most from purpose-built tools.

E SKILL.md Format Specification

```
---
name: <skill-identifier>
```

```
version: 1.0.0
trigger: >
  <Natural language description of when this skill applies.
   Should be specific enough to avoid false positives.>
scope:
  - developer
  - operator       # optional: role-specific scoping
tags:
  - <tag1>
  - <tag2>
difficulty: beginner | intermediate | advanced
estimated_time: <human-readable duration>
---

# <Skill Title>

## Overview

<One-paragraph description of what this skill accomplishes.>

## When to Use

<Bullet list of triggering conditions.>

## Procedure

1. <Step 1>
2. <Step 2>
   ...

## Common Pitfalls

- <Pitfall 1 and how to avoid it>
- <Pitfall 2 and how to avoid it>

## Examples

### Example 1: <Short description>

<Concrete example with code snippets if applicable.>

## See Also

- <Related skill name>
- <Related extension name>
```

References

Anthropic. Claude 3.5 technical report. Technical report, Anthropic, 2024.

Anthropic. Model Context Protocol: Open standard for context in AI systems. https://modelcontextprotocol.io, 2024.

Bavishi, R., Lemieux, C., Fox, R., Sen, K., and Stoica, I. AutoPandas: Neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA): 1–27, 2019.

Claessen, K. and Hughes, J. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 268–279, 2000.

Git. git-worktree: Manage multiple working trees. https://git-scm.com/docs/git-worktree, 2015.

Gulwani, S., Polozov, O., and Singh, R. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1–2): 1–119, 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.

Holland, J. H. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

Hu, S., Lu, C., and Clune, J. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

Koza, J. R. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.

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

ngrok. bmo: An agent that builds its own tools. https://github.com/ngrok/bmo, 2024.

OpenAI. GPT-4 technical report. Technical report, OpenAI, 2023.

Schmidhuber, J. Evolutionary principles in self-referential learning. Diploma thesis, Technische Universität München, 1987.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y., Wu, Y., and Guo, D. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Sobania, D., Briesch, M., Hanna, C., and Petke, J. An analysis of the automatic bug fixing performance of ChatGPT. In *IEEE/ACM International Workshop on Automated Program Repair (APR)*, pp. 23–30, 2023.

Thrun, S. and Pratt, L. *Learning to Learn*. Springer, 1998.

Wang, Y., Kordi, Y., Mishra, S., Liu, A., Smith, N. A., Khashabi, D., and Hajishirzi, H. Self-Instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 13484–13508, 2023.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Yuan, Z., Yuan, H., Li, C., Dong, G., Lu, K., Tan, C., Zhou, C., and Zhou, J. Scaling relationship on learning mathematical reasoning with large language models. *arXiv preprint arXiv:2308.01825*, 2023.

Zelikman, E., Wu, Y., Mu, J., and Goodman, N. D. STaR: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.