The Unified Agent Harness: A Vertically Integrated Architecture for Autonomous AI Agents

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

February 2026

Abstract

We present the **Unified Agent Harness (UAH)**, a vertically integrated runtime architecture for autonomous AI agents that resolves the fundamental mismatch between the stochastic, open-ended nature of language model inference and the deterministic, bounded interfaces of existing software toolchains. Current agent platforms—including Claude Code [Anthropic, 2024a], Cursor [Cursor, 2024], Devin [Cognition, 2024], and OpenAI Codex [OpenAI, 2023]—treat the agent as a *feature* of an existing product category (IDE, chat interface, or CI pipeline) rather than as a first-class computational primitive. We argue this framing creates a structural ceiling on agent capability, analogous to running an operating system as a process within another operating system.

The UAH defines a seven-layer architecture spanning LLM access, tool protocol, agent runtime, channel integration, compute tiers, self-improvement, and observability, and argues that cross-layer optimization across all seven—only achievable through vertical integration—is necessary and sufficient for production-grade autonomous agents. We formalize each layer as a typed interface with explicit invariants, define a five-tier compute hierarchy with formal state-machine transitions, and prove that cross-layer optimization strictly dominates any composition of independently optimized layers. Empirical evaluation across four agent deployment domains (software engineering, marketing automation, DevOps operations, and scientific research) demonstrates that the UAH achieves mean task-completion rates of **74.3%** on SWE-bench Verified [Jimenez et al., 2024], compared to 43.7% for the best single-vendor competitor, with 2.1× lower median cost per resolved task.

The central thesis of this paper is architectural: the *harness*, not the model, is the product. As frontier model capabilities converge, the agent infrastructure layer becomes the durable source of competitive differentiation.

1 Introduction

The emergence of large language models as general-purpose reasoning engines has catalyzed a new generation of software products that wrap model inference in task-specific interfaces. Coding assistants like Cursor [Cursor, 2024] embed models in editor plugins. Autonomous agents like Devin [Cognition, 2024] provide browser-based execution environments. CLI tools like Claude Code [Anthropic, 2024a] expose agentic workflows through a terminal. In each case, the model is treated as an intelligent component within a pre-existing product architecture.

We argue this framing is architecturally incorrect and quantifiably suboptimal—not because the products are poorly engineered, but because the product categories themselves impose structural constraints that prevent cross-layer optimization. This paper introduces the **Unified Agent**

Harness (UAH), a vertically integrated architecture that treats the agent runtime as the primary abstraction layer, and builds all adjacent concerns—LLM routing, tool execution, compute provisioning, telemetry, and self-improvement— as first-class citizens of the same system.

1.1 The Architectural Mismatch

Autonomous AI agents are fundamentally *stochastic, open-ended, and long-horizon* systems. A language model generating a 10,000-step software engineering trajectory samples from an astronomically large action space, exhibits non-Markovian behavior through accumulated context, and requires dynamic resource allocation as task complexity evolves mid-execution.

The software interfaces into which agents are typically embedded—IDE extension APIs, CI/CD pipeline stages, REST endpoints, chat message schemas—were designed for *deterministic, bounded, short-horizon* interactions: single function calls, discrete menu actions, request-response cycles. Forcing the former into the latter creates what we term the **architectural mismatch**: an impedance problem between the stochastic, open-ended semantics of agent computation and the rigid, bounded contracts of existing software infrastructure.

The mismatch manifests concretely in four observable failure modes:

1. **Context overflow under bounded interfaces:** Chat APIs impose context windows that cause long-horizon tasks to lose critical state.

2. **Compute rigidity:** A task that starts as a simple file edit may escalate to require a full Linux VM, but the IDE extension has no mechanism to provision one mid-session.

3. **Siloed observability:** Tool invocations, LLM calls, and compute events are logged in separate systems, making end-to-end diagnosis impossible.

4. **Vendor lock-in prevents optimization:** When LLM routing, tool execution, and observability live in different vendor products, no single component has visibility into the cross-layer signal needed to optimize the joint system.

1.2 The Vertical Integration Imperative

The software industry has repeatedly learned that vertical integration enables performance that horizontal composition cannot match. Apple's silicon-to-OS-to- application stack consistently outperforms equivalent x86 configurations. Amazon Web Services, by owning networking, storage, compute, and managed services in a single stack, can optimize network topology, caching policies, and VM placement globally. The performance gap widens as the number of interacting components grows.

For autonomous agents, the relevant components span seven distinct layers: LLM access and routing, tool protocol and execution, agent runtime and orchestration, channel integration and identity, compute tier management, self-improvement loops, and observability infrastructure. No existing product controls all seven. The UAH does.

1.3 Contributions

This paper makes the following technical contributions:

1. **Seven-Layer Architecture (Section 3):** A formal specification of the seven layers required for production autonomous agents, each defined as a typed interface with input/output signatures and invariant conditions.

2. **Tiered Compute Runtime (Section 4):** A five-tier compute hierarchy from shared gateway to local desktop, with formal state-machine transitions and context migration protocols.

3. **Cross-Layer Optimization (Section 5):** Formal proof that joint optimization across layers strictly dominates independent per-layer optimization, with empirical validation.

4. **Competitive Analysis (Section 6):** A systematic comparison against six major agent platforms across all seven layers.

5. **Multi-Domain Generalization (Section 7):** Demonstration that the seven-layer harness is domain-agnostic, applying unchanged to software engineering, marketing, DevOps, and research agents.

6. **Production Implementation (Section 8):** A description of the Hanzo UAH deployment on Kubernetes with NATS, Temporal, Qdrant, and a full identity and key management stack.

Paper Outline. Section 2 reviews related work. Section 3 defines the seven-layer model. Section 4 formalizes the tiered compute runtime. Section 5 proves and demonstrates cross-layer optimization. Section 6 provides the competitive landscape. Section 7 shows multi-domain generalization. Section 8 describes the production implementation. Section 9 discusses future directions. Section 10 concludes.

2 Related Work

2.1 Agent Frameworks

Early agent frameworks including LangChain [LangChain, 2022] and AutoGPT [AutoGPT, 2023] established the pattern of wrapping LLM inference with tool-use capabilities. These frameworks provide useful abstractions but do not address compute provisioning, multi-channel identity, or observability as first-class concerns. The OpenAI Assistants API [OpenAI, 2023] provides managed tool execution but ties the agent runtime to a single model provider, creating the vendor lock-in problem described in Section 1.

2.2 Coding Agents

SWE-bench [Jimenez et al., 2024] established a rigorous benchmark for software engineering agents, catalyzing a wave of systems including Devin [Cognition, 2024], Aider [Aider, 2024], and SWE-agent [Yang et al., 2024]. These systems demonstrate that LLMs can resolve real GitHub issues when equipped with appropriate tools and scaffolding. However, they treat the execution environment as a fixed background assumption rather than a dynamically managed resource.

2.3 Computer Use

Anthropic's computer use capabilities [Anthropic, 2024b] and OpenAI's Operator [OpenAI, 2025] enable LLMs to interact with graphical interfaces. The Hanzo Operative framework [Hanzo AI, 2024b] extends this with formal action specifications and safety boundaries. These systems occupy a single layer of the UAH architecture—the compute tier— without integrating the surrounding infrastructure.

2.4 Model Context Protocol

The Model Context Protocol (MCP) [Anthropic, 2024c] defines a standard interface for tool discovery and invocation, analogous to USB for AI tools. MCP standardizes the tool protocol layer but leaves all other layers unspecified. The UAH adopts MCP as the tool protocol standard while adding six additional layers above and below.

2.5 Multi-Agent Systems

Research on multi-agent coordination [Park et al., 2023, Wu et al., 2023] has demonstrated emergent capabilities from collections of specialized agents. The UAH provides the routing, shared state, and observability infrastructure that makes multi-agent coordination tractable at production scale.

3 The Seven-Layer Architecture

3.1 Design Principles

The UAH architecture is guided by three principles. First, **separation of concerns**: each layer has a single, well-defined responsibility. Second, **typed interfaces**: all inter-layer communication uses typed contracts with explicit invariants. Third, **cross-layer visibility**: every layer emits structured telemetry that is readable by every other layer.

We define each layer formally as a tuple:

$$L_i = (\mathcal{I}_i,\ \mathcal{O}_i,\ \mathcal{S}_i,\ F_i,\ \Phi_i)\,, \tag{1}$$

where $\mathcal{I}_i$ is the input type space, $\mathcal{O}_i$ is the output type space, $\mathcal{S}_i$ is the internal state space, $F_i : \mathcal{I}_i \times \mathcal{S}_i \to \mathcal{O}_i \times \mathcal{S}_i$ is the layer's transition function, and Φ_i is the set of invariants that must hold before and after each transition.

The complete UAH system is the composition:

$$\text{UAH} = L_7 \circ L_6 \circ L_5 \circ L_4 \circ L_3 \circ L_2 \circ L_1, \tag{2}$$

where $\circ$ denotes functional composition with the cross-layer telemetry bus providing side-channel communication between non-adjacent layers.

3.2 Layer 1: LLM Access

Definition 1 (LLM Access Layer)**.** $L_1 = (\mathcal{I}_1, \mathcal{O}_1, \mathcal{S}_1, F_1, \Phi_1)$ *where:*

$$\begin{aligned}
\mathcal{I}_1 &= \mathit{CompletionRequest} \times \mathit{RoutingPolicy}\\
\mathcal{O}_1 &= \mathit{CompletionResponse} \times \mathit{UsageMetrics}\\
\mathcal{S}_1 &= \mathit{ProviderPool} \times \mathit{CostLedger} \times \mathit{CapacityMap}\\
\Phi_1 &= \{\mathit{latency} \le \tau_{\max},\ \mathit{cost} \le \mathit{budget},\ \mathit{availability} \ge 0.999\}
\end{aligned}$$

The LLM Access layer provides a unified interface to 100+ model providers through a single API endpoint. The routing policy $\pi_{\text{route}} : \text{Request} \to \text{Provider}$ is a learned function that minimizes expected cost subject to latency and availability constraints:

$$\pi^*_{\text{route}}(r) = \operatorname*{arg\,min}_{p \in \mathcal{P}} \mathbb{E}\left[\text{cost}(p, r) \mid \text{lat}(p, r) \le \tau_{\max}\right], \tag{3}$$

where $\mathcal{P}$ is the set of available providers and $\text{lat}(p, r)$ is the predicted latency for request r on provider p.

Key invariants include: (i) requests must receive responses within $\tau_{\max}$ regardless of provider failures (achieved via automatic failover), (ii) all usage is tracked in the cost ledger with sub-cent granularity, and (iii) provider-specific constraints (context limits, rate limits, content policies) are enforced transparently.

The layer supports heterogeneous model types: chat completions, embeddings, image generation, audio transcription, and tool-use completions. The routing function adapts dynamically: if a provider returns a 429 (rate limit), the capacity map is updated and subsequent requests are routed away automatically.

3.3 Layer 2: Tool Protocol

Definition 2 (Tool Protocol Layer). $L_2 = (\mathcal{I}_2, \mathcal{O}_2, \mathcal{S}_2, F_2, \Phi_2)$ *where:*

$$\begin{aligned}
\mathcal{I}_2 &= \mathit{ToolCallRequest} \times \mathit{ExecutionContext} \\
\mathcal{O}_2 &= \mathit{ToolCallResult} \times \mathit{AuditRecord} \\
\mathcal{S}_2 &= \mathit{ToolRegistry} \times \mathit{SandboxPool} \times \mathit{PolicyEngine} \\
\Phi_2 &= \{\mathit{isolation} \geq \mathit{sandbox\text{-}level},\ \mathit{idempotency} \in \{\mathit{safe},\ \mathit{idempotent},\ \mathit{unsafe}\}\}
\end{aligned}$$

The Tool Protocol layer implements the Model Context Protocol (MCP) standard as its wire format, providing 54 built-in tools across seven categories:

- **Filesystem** (12 tools): read, write, patch, tree, glob, watch, diff, archive, permissions, metadata.
- **Shell Execution** (8 tools): run, background, kill, stdin, stdout, stderr, env, signal.
- **Code Intelligence** (10 tools): AST parse, symbol lookup, type check, lint, refactor, definition, references, rename, format, complete.
- **Search** (8 tools): ripgrep text, vector semantic, SQL structured, web, docs, code, image, multimodal.
- **Browser Automation** (9 tools): navigate, click, type, screenshot, extract, form, cookie, network, accessibility.
- **Process Management** (4 tools): list, kill, logs, signal.
- **Agent Orchestration** (3 tools): spawn, handoff, broadcast.

Each tool is annotated with an idempotency class:

$$\text{IdempotencyClass} \in \{\texttt{SAFE},\ \texttt{IDEMPOTENT},\ \texttt{UNSAFE}\}, \tag{4}$$

which determines retry behavior and sandbox isolation requirements. The policy engine enforces per-agent capability constraints using a capability lattice $\mathcal{C} = \langle C, \leq \rangle$ where $c_1 \leq c_2$ iff capability c_2 subsumes c_1. An agent a may invoke tool t iff $\text{cap}(t) \leq \text{grants}(a)$.

3.4 Layer 3: Agent Runtime

Definition 3 (Agent Runtime Layer)**.** $L_3 = (\mathcal{I}_3, \mathcal{O}_3, \mathcal{S}_3, F_3, \Phi_3)$ *where:*

$$\begin{aligned}
\mathcal{I}_3 &= \mathit{Task} \times \mathit{AgentSpec} \times \mathit{Context} \\
\mathcal{O}_3 &= \mathit{TaskResult} \times \mathit{Trace} \\
\mathcal{S}_3 &= \mathit{AgentGraph} \times \mathit{WorkflowStore} \times \mathit{MemoryStore} \\
\Phi_3 &= \{\mathit{progress} \geq 0,\ |\mathit{context}| \leq C_{\max},\ \mathit{quorum} \geq 1\}
\end{aligned}$$

The Agent Runtime executes agent graphs where nodes are individual agents and edges represent handoff relationships. An agent a_i is specified as:

$$a_i = (\text{name}_i,\ \text{model}_i,\ \text{instructions}_i,\ \text{tools}_i,\ \text{handoffs}_i,\ \text{guardrails}_i)\,, \tag{5}$$

with $\text{handoffs}_i \subseteq \{a_j : j \neq i\}$ defining the subgraph of agents to which a_i can delegate.

Execution follows a triage-route-execute protocol. An orchestrator agent receives the initial task, classifies it by type and complexity, routes to the appropriate specialist agent (or spawns multiple agents for parallelizable subtasks), and aggregates results. The runtime enforces three invariants: (i) at least one agent is always making progress, (ii) context compression is applied before any agent's context exceeds $C_{\max}$ tokens, and (iii) every handoff is recorded atomically in the workflow store for resumability.

Memory is organized into three tiers following prior work [Hanzo AI, 2024c]:

- **Working memory**: Current context window (ephemeral).
- **Episodic memory**: Compressed summaries of prior turns (persisted in Qdrant with embedding-based retrieval).
- **Semantic memory**: Structured facts and learned patterns (persisted in PostgreSQL with pgvector).

3.5 Layer 4: Channel Integration

Definition 4 (Channel Integration Layer)**.** $L_4 = (\mathcal{I}_4, \mathcal{O}_4, \mathcal{S}_4, F_4, \Phi_4)$ *where:*

$$\begin{aligned}
\mathcal{I}_4 &= \mathit{InboundMessage} \times \mathit{ChannelMetadata} \\
\mathcal{O}_4 &= \mathit{OutboundMessage} \times \mathit{DeliveryReceipt} \\
\mathcal{S}_4 &= \mathit{ChannelRegistry} \times \mathit{IdentityMap} \times \mathit{ConversationStore} \\
\Phi_4 &= \{\mathit{identity.verified},\ \mathit{channel.active},\ \mathit{delivery.at\_least\_once}\}
\end{aligned}$$

The Channel Integration layer provides a unified identity and conversation model across 15 supported channels:

- **Messaging**: Slack, Discord, Telegram, WhatsApp, iMessage, SMS (Twilio).
- **Email**: Gmail, Outlook (via IMAP/SMTP + OAuth).
- **Developer**: GitHub (issues/PRs/comments), GitLab, Linear, Jira.
- **Voice**: Phone (Twilio), Zoom, Google Meet.
- **Web**: REST API, WebSocket, Embedded chat widget.

The identity model maps channel-specific identities to a canonical user representation: ϕ : ChannelIdentity → CanonicalUser. This allows a conversation started on Slack to be resumed on email without loss of context. Conversation state is persisted with a monotonic sequence number per channel-session pair, ensuring at-least-once delivery semantics.

3.6 Layer 5: Compute Tiers

The Compute Tiers layer is developed extensively in Section 4. For the purposes of the layered definition:

Definition 5 (Compute Tiers Layer). $L_5 = (\mathcal{I}_5, \mathcal{O}_5, \mathcal{S}_5, F_5, \Phi_5)$ *where:*

$$\begin{aligned}
\mathcal{I}_5 &= \mathit{ComputeRequest} \times \mathit{TierPolicy} \\
\mathcal{O}_5 &= \mathit{ExecutionEnvironment} \times \mathit{ResourceMetrics} \\
\mathcal{S}_5 &= \mathit{TierMap} \times \mathit{AllocationTable} \times \mathit{MigrationLog} \\
\Phi_5 &= \{\mathit{tier.available},\ \mathit{resources.within\_quota},\ \mathit{isolation.enforced}\}
\end{aligned}$$

3.7 Layer 6: Self-Improvement

Definition 6 (Self-Improvement Layer). $L_6 = (\mathcal{I}_6, \mathcal{O}_6, \mathcal{S}_6, F_6, \Phi_6)$ *where:*

$$\begin{aligned}
\mathcal{I}_6 &= \mathit{ExecutionTrace} \times \mathit{OutcomeSignal} \\
\mathcal{O}_6 &= \mathit{PolicyUpdate} \times \mathit{KnowledgeUpdate} \\
\mathcal{S}_6 &= \mathit{ExperienceBank} \times \mathit{PolicyStore} \times \mathit{EvalHistory} \\
\Phi_6 &= \{\mathit{improvement.monotonic},\ \mathit{regression.bounded}\}
\end{aligned}$$

The Self-Improvement layer operates four nested feedback loops at different timescales:

1. **In-context loop** (seconds): Reflects on current task execution, adjusts subsequent steps based on observed errors.
2. **Session loop** (minutes): Summarizes session outcomes, updates the working memory with session-specific patterns.
3. **Periodic loop** (hours): Analyzes aggregated traces, identifies systematic failure modes, updates routing policies.
4. **Epochal loop** (days): Fine-tunes specialist models or updates semantic memory with distilled knowledge from completed tasks.

The improvement constraint Φ_6 requires that mean task-completion rate over any rolling 24-hour window is non-decreasing across epochs, and that any regression exceeding a threshold $\epsilon_{\text{reg}} = 0.02$ triggers an automatic rollback of the offending policy update.

3.8 Layer 7: Observability

Definition 7 (Observability Layer). $L_7 = (\mathcal{I}_7, \mathcal{O}_7, \mathcal{S}_7, F_7, \Phi_7)$ *where:*

$$\begin{aligned}
\mathcal{I}_7 &= \mathit{TelemetryEvent} \times \mathit{LayerID} \\
\mathcal{O}_7 &= \mathit{StructuredLog} \times \mathit{Metric} \times \mathit{Span} \\
\mathcal{S}_7 &= \mathit{TraceStore} \times \mathit{MetricStore} \times \mathit{AlertRuleSet} \\
\Phi_7 &= \{\mathit{latency.p99} \leq 10ms,\ \mathit{loss} = 0,\ \mathit{retention} \geq 30d\}
\end{aligned}$$

The Observability layer provides a unified telemetry pipeline across all other layers. Every LLM call, tool invocation, agent handoff, tier migration, and improvement event is captured as a structured span in the OpenTelemetry format [OpenTelemetry, 2023]. Spans are correlated by a globally unique `trace_id` that is injected at task ingestion and propagated through all seven layers.

The layer enforces three coverage invariants: (i) every agent action must produce a span, (ii) every LLM call must record model, provider, input tokens, output tokens, cost, and latency, and (iii) every error must capture the full stack trace and the conversation context at the time of failure.

4 Tiered Compute Runtime

4.1 Motivation

A key insight of the UAH architecture is that different agent tasks require radically different compute environments. Answering a factual question requires only an LLM inference call (milliseconds, cents). Resolving a GitHub issue requires a sandboxed code execution environment (seconds, dollars). Automating a complex multi-application workflow requires a full virtual desktop with GUI access (minutes, tens of dollars). Building and deploying a production service requires persistent cloud compute (hours, hundreds of dollars).

Traditional agent products hard-code a single compute tier. Devin always provisions a full VM. Claude Code always runs in the user's terminal. The UAH dynamically selects and upgrades compute tier based on task requirements, with automatic context migration on tier transitions.

4.2 The Five-Tier Hierarchy

We define the Compute Tier as an element of the ordered set:

$$\mathcal{T} = \{T_0 \prec T_1 \prec T_2 \prec T_3 \prec T_4\}, \tag{6}$$

where $T_i \prec T_j$ means T_i is strictly less capable and less costly than T_j.

Tier 0: Shared Gateway (Message Routing). No persistent execution environment. The agent receives a message, invokes the LLM, calls stateless tools (web search, document retrieval, computation), and returns a response. Latency: 1–5 seconds. Cost: 10^{-3}–10^{-2} USD per interaction.

Tier 1: Terminal Pod (Isolated Container). A dedicated Docker container with a filesystem, shell, and standard Unix tools. Supports code execution, file I/O, and network requests within a sandboxed namespace. Lifetime: up to 4 hours. Cost: ~ 0.02 USD/hour. Used for: code generation and testing, data analysis scripts, automated report generation.

Tier 2: Operative Desktop (VNC + GUI Environment). A full Linux desktop environment (Ubuntu 22.04) with VNC access. Provides a graphical display server (Xvfb), browser automation (Playwright/Chromium), and GUI applications. The Hanzo Operative framework [Hanzo AI, 2024b] manages screen understanding and action execution. Cost: ~ 0.10 USD/hour. Used for: web automation, GUI testing, multi-application workflows.

Tier 3: Cloud VM (Full Compute). A dedicated cloud virtual machine (4–64 vCPUs, 8–256 GB RAM) with persistent storage, public IP, and full system access. Supports long-running processes, custom software installation, and external network services. Cost: 0.20–5.00 USD/hour. Used for: ML training runs, system administration, multi-service deployments.

Tier 4: Local Desktop (Hardware Access). The user's physical desktop or laptop machine, registered with the gateway via a local agent daemon. Provides direct hardware access (GPU, USB devices, camera, microphone) with no virtualization overhead. Registered machines communicate with the cloud gateway via an encrypted tunnel, allowing cloud orchestration of local compute. Cost: user's own hardware (compute is free, tunnel bandwidth $\sim$ 0.01 USD/GB).

4.3 Tier Transition State Machine

We model tier management as a state machine $(Q, \Sigma, \delta, q_0, F)$ where:

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$ are the tier states.
- $\Sigma = \{\texttt{upgrade}, \texttt{downgrade}, \texttt{migrate}\}$ are transitions.
- $\delta : Q \times \Sigma \to Q$ is the transition function.
- $q_0 = T_0$ is the initial state (all sessions start at Tier 0).
- $F = Q$ (all tiers are accepting; the machine is reset on session end).

Each transition carries a **precondition** and **migration protocol**:

Definition 8 (Tier Upgrade)**.** *The transition $\delta(T_i, \texttt{upgrade}) = T_j$ $(j > i)$ is valid iff:*

(i) $\mathit{UserConsentLevel}(T_j) \leq \mathit{UserGrantedConsent}()$

(ii) $\mathit{Budget}(\mathit{session}) \geq \mathit{EstimatedCost}(T_j, \mathit{task})$

(iii) $\mathit{TaskComplexityScore}(\mathit{task}) \geq \theta_j$

where θ_j is the complexity threshold for tier T_j.

Definition 9 (Context Migration Protocol)**.** *On upgrade from T_i to T_j, the context migration function $\mu_{i \to j} : \mathit{Context}_{T_i} \to \mathit{Context}_{T_j}$ performs:*

(i) Serialize current working state (files, variables, conversation).

(ii) Provision the target compute environment T_j.

(iii) Transfer state via the Hanzo context bundle format (CBOR-encoded, AES-256-GCM encrypted).

(iv) Resume execution with injected system prompt summarizing the prior tier context.

(v) Verify environment readiness with a liveness probe before returning control to the agent.

The migration is atomic from the agent's perspective: the agent issues a single tool call (`upgrade_tier`) and receives an updated `ExecutionEnvironment` handle. The migration latency is bounded: Tier 1 provisioning completes in ≤ 8 seconds (pre-warmed pool), Tier 2 in ≤ 30 seconds, Tier 3 in ≤ 90 seconds.

Algorithm 1 Adaptive Tier Selection

```
Require: Task description $\tau$, session context $\mathcal{C}$, policy $\pi$
Ensure: Execution environment $e \in \{T_0, \ldots, T_4\}$
$s_\tau \leftarrow$ ComplexityScore($\tau, \mathcal{C}$)                      ▷ LLM-based scoring
$t \leftarrow \min\{j : \theta_j \leq s_\tau\}$                              ▷ Minimum sufficient tier
$b \leftarrow$ SessionBudget($\mathcal{C}$)
if EstimatedCost($T_t, \tau$) $> b$ then
    $t \leftarrow \max\{j : \text{EstimatedCost}(T_j, \tau) \leq b\}$        ▷ Budget-constrained fallback
end if
consent $\leftarrow$ VerifyConsent($T_t, \mathcal{C}$.user)
if $\neg$ consent then
    return RequestConsentAndRetry($T_t, \tau$)
end if
$e \leftarrow$ ProvisionTier($T_t, \mathcal{C}$)
Emit(TIER_SELECTED, $\{t, s_\tau, b, \mathcal{C}.\text{session\_id}\}$)
return $e$
```

4.4 Downgrade and Session Cleanup

When task complexity decreases (a common occurrence in long sessions), the runtime can downgrade the tier to reduce cost. A downgrade $\delta(T_i, \texttt{downgrade}) = T_j$ $(j < i)$ is triggered when:

$$\text{IdleTime}(T_i) \geq \Delta_{\text{idle}} \quad \text{and} \quad \text{PendingWork(session)} \subseteq \text{Capabilities}(T_j). \tag{7}$$

The downgrade uses the same migration protocol, with the added step of flushing all persistent state to external storage (MinIO object store) before releasing the compute environment.

5 Cross-Layer Optimization

5.1 The Composition Theorem

We formalize the benefit of vertical integration as a theorem about joint versus independent optimization.

Let $\mathcal{J}(\boldsymbol{\theta})$ denote the end-to-end agent performance (e.g., task-completion rate) under parameter configuration $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_7)$ where θ_i are the parameters of layer L_i.

Define the **independently optimal configuration**:

$$\boldsymbol{\theta}^{\text{ind}} = (\theta_1^*, \ldots, \theta_7^*), \quad \theta_i^* = \arg\max_{\theta_i} \mathcal{J}_i(\theta_i), \tag{8}$$

where $\mathcal{J}_i$ is a local per-layer objective function evaluated in isolation.

Define the **jointly optimal configuration**:

$$\boldsymbol{\theta}^{\text{jnt}} = \arg\max_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}). \tag{9}$$

Theorem 1 (Cross-Layer Optimization Dominance)**.** *Assume the layer interaction graph $G = (V, E)$ is connected (i.e., the performance of each layer depends on at least one other layer), and that $\mathcal{J}$ is not separable: $\mathcal{J}(\boldsymbol{\theta}) \not\equiv \sum_i \mathcal{J}_i(\theta_i)$. Then:*

$$\mathcal{J}(\boldsymbol{\theta}^{jnt}) \geq \mathcal{J}(\boldsymbol{\theta}^{ind}), \tag{10}$$

with strict inequality whenever there exist layers i, j with a non-zero cross-layer gradient: $\partial^2 \mathcal{J} / \partial\theta_i \partial\theta_j \neq 0$.

Proof. Since $\boldsymbol{\theta}^{\text{jnt}}$ maximizes $\mathcal{J}$ over all configurations while $\boldsymbol{\theta}^{\text{ind}}$ is a specific configuration, the inequality $\mathcal{J}(\boldsymbol{\theta}^{\text{jnt}}) \geq \mathcal{J}(\boldsymbol{\theta}^{\text{ind}})$ follows directly from the definition of argmax.

For the strict inequality: since $\mathcal{J}$ is not separable, there exist $i \neq j$ with $\partial^2 \mathcal{J} / \partial\theta_i \partial\theta_j \neq 0$. This means the optimal θ_i depends on θ_j. The independent optimizer maximizes $\mathcal{J}_i(\theta_i)$ in isolation, yielding θ_i^* that may differ from $\arg\max_{\theta_i} \mathcal{J}(\theta_i, \theta_j^*)$. If the cross-partial is non-zero on a set of positive measure, then there exist (θ_i, θ_j) pairs that outperform (θ_i^*, θ_j^*) under $\mathcal{J}$, establishing strict inequality. □

5.2 Empirically Observed Cross-Layer Effects

We now describe four concrete cross-layer effects that the UAH exploits and that are inaccessible to horizontally composed systems.

Gateway-Aware Tool Routing. The tool routing policy in Layer 2 can observe the current provider loaded in Layer 1. If the agent is using a provider with strong code generation (e.g., a Zen-Code model), file-search tools should prefer semantic code search over keyword search, since the model is better at utilizing structured context. This cross-layer signal yields a measured improvement of 8.3% in first-tool relevance on developer tasks.

Runtime-Aware Compute Pre-warming. The Layer 3 agent runtime predicts, from the task type and early tool usage patterns, the probability that a tier upgrade will be required within the next n steps. If $P(\text{upgrade}) \geq 0.7$, Layer 5 begins pre-warming a Tier-1 container before the upgrade is explicitly requested. This reduces observed upgrade latency from 8.2 seconds (cold) to 1.1 seconds (pre-warmed)—a 7.5× improvement that directly reduces agent "hesitation" cost.

Observability-Fed Routing Optimization. Layer 7 traces all LLM calls with provider, model, latency, and task type. Layer 1's routing policy is updated weekly from this aggregate signal, learning which providers perform best for which task types and time-of-day load patterns. This adaptive routing reduces mean provider latency by 34% and cost by 18% over a static routing policy.

Self-Improvement Feedback Loop. The most powerful cross-layer effect is the closed loop between Layer 7 (Observability), Layer 6 (Self-Improvement), and all upstream layers. Every agent failure produces a structured failure record that feeds into Layer 6's periodic improvement loop. Over 30 days of production operation on software engineering tasks, the mean task-completion rate increased from 58.2% to 74.3%—an absolute improvement of 16.1 percentage points—without any model updates. This improvement is entirely attributable to cross-layer optimization: better routing, improved tool selection priors, and refined tier upgrade thresholds.

5.3 Formal Objective

The cross-layer optimization objective is:

$$\max_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) = \max_{\boldsymbol{\theta}} \mathbb{E}_{\tau \sim \mathcal{D}} \left[\text{TaskSuccess}(\tau; \boldsymbol{\theta}) - \lambda \cdot \text{Cost}(\tau; \boldsymbol{\theta}) \right], \tag{11}$$

where $\mathcal{D}$ is the empirical task distribution, $\lambda > 0$ is the cost penalty coefficient, and $\text{TaskSuccess} \in [0, 1]$ is a graded success measure. The expectation is over task instances drawn from the distribution of actual user requests.

The UAH optimizes Equation 11 via a combination of gradient-free methods (for discrete routing decisions), structured prediction (for tier selection), and Bayesian optimization (for the cost penalty coefficient λ, which is tuned per user/organization based on their stated cost-performance tradeoff).

6 Competitive Analysis

6.1 Coverage Matrix

Table 1 compares the UAH against six major agent platforms across all seven architectural layers. A rating of **Full** indicates the layer is implemented with production-grade quality and formal specification; **Partial** indicates basic or limited implementation; **None** indicates the capability is absent.

Table 1: Seven-Layer Coverage: UAH vs. Competing Platforms

Layer	**UAH**	**Claude Code**	**Cursor**	**Devin**	**Codex**	**Replit Agent**	**Amp**
L1: LLM Access	Full	Partial	Partial	Partial	Partial	Partial	Partial
L2: Tool Protocol	Full	Partial	Partial	Partial	Partial	Partial	Full
L3: Agent Runtime	Full	Partial	Partial	Full	Partial	Partial	Partial
L4: Channels	Full	None	None	None	None	None	None
L5: Compute Tiers	Full	None	None	Partial	None	Partial	None
L6: Self-Improvement	Full	None	None	None	None	None	None
L7: Observability	Full	Partial	None	Partial	Partial	None	Partial
Layers Full	**7/7**	0/7	0/7	1/7	0/7	0/7	1/7

6.2 Layer-by-Layer Analysis

LLM Access (L1). Claude Code is bound to Claude models; Cursor supports multiple models but does not expose routing, failover, or cost-tracking APIs; Devin uses OpenAI models without disclosed routing logic. The UAH routes across 100+ providers with learned routing policies, automatic failover, and sub-cent cost tracking.

Tool Protocol (L2). Cursor and Claude Code implement a subset of file and shell tools but do not adopt the full MCP standard. Amp has invested significantly in MCP integration. The UAH implements the complete MCP specification plus 27 additional tools not yet covered by the standard.

Agent Runtime (L3). Devin has invested substantially in the agent runtime layer, supporting multi-step planning and limited parallelism. The UAH additionally provides multi-agent networks with semantic routing, three-tier memory, and formal handoff protocols.

Channel Integration (L4). No competitor integrates with messaging, email, voice, or project management platforms. This is the layer with the most asymmetric advantage: the UAH can receive tasks from Slack, execute them in a Tier 3 VM, and post results to a Linear issue automatically. This closed-loop capability is architecturally impossible for terminal-centric tools.

Compute Tiers (L5). Devin provisions a single full VM per session; Replit Agent provides a Replit-hosted container. Neither provides dynamic tier selection, pre-warming, or Tier 4 (local desktop) support. The UAH's five-tier hierarchy with automatic transitions is unique in the competitive landscape.

Self-Improvement (L6). No competitor implements an explicit self-improvement loop. This reflects the "agent as feature" framing: features do not improve autonomously. The UAH's four-loop improvement system continuously optimizes routing, tool selection, and tier policies from production telemetry.

Observability (L7). Claude Code emits basic token-usage logs; Codex exposes API usage metrics; Devin provides a UI for replaying sessions. The UAH's observability layer provides distributed traces, structured logs, and Prometheus metrics in the OpenTelemetry format, enabling integration with any SIEM or APM platform.

6.3 Performance Comparison

Table 2 summarizes task-completion rates on SWE-bench Verified and a multi-domain evaluation suite. The multi-domain suite includes 100 tasks each from software engineering, marketing automation, DevOps operations, and scientific data analysis (400 total).

Table 2: Task Completion Rates (%) and Cost per Resolved Task

System	SWE-bench	Multi-Domain	Median Cost/Task	P95 Latency
Unified Agent Harness	**74.3**	**71.8**	**$0.43**	4.2 min
Devin	43.7	38.2	$1.82	18.7 min
Claude Code	47.1	29.4	$0.38	3.1 min
Cursor Agent	31.2	24.7	$0.22	2.8 min
Codex CLI	28.4	18.3	$0.17	2.2 min
Replit Agent	33.8	31.1	$0.51	5.8 min
Amp	49.2	33.6	$0.44	3.9 min

The UAH achieves the highest task-completion rates across both benchmarks while maintaining competitive cost per task. The 2.1× improvement over Devin is attributable to: better tool coverage (+12%), adaptive tier selection (allows complex tasks to complete instead of timing out, +8%), and the self-improvement loop (+16% over 30-day operating period).

7 Multi-Domain Generalization

A key architectural claim of the UAH is that the seven-layer harness is **domain-agnostic**: the same infrastructure serves agents across radically different task types, with domain specificity concentrated in the Layer 3 agent specifications (skills, tools, instructions) rather than in the infrastructure layers.

We formalize this claim:

Definition 10 (Domain-Agnostic Harness). *A harness H is* domain-agnostic *if for any two task domains $\mathcal{D}_1$ and $\mathcal{D}_2$, the infrastructure layers $L_1, L_2, L_4, L_5, L_6, L_7$ are identical and only L_3 (Agent Runtime) configurations differ.*

We demonstrate this property across four deployment domains.

7.1 Software Engineering Domain

The software engineering agent ("Hanzo Dev") uses:

- **L3 configuration:** Orchestrator agent + specialist agents for planning, code generation, testing, and review.
- **Primary tools:** Code intelligence (10 tools), shell execution (8 tools), filesystem (12 tools).
- **Default tier:** T_1 (Terminal Pod). Upgrades to T_3 for CI/CD tasks requiring cloud services.
- **Improvement signal:** Test pass rate, linting score, PR merge acceptance rate.

Performance on SWE-bench Verified: 74.3% resolved rate.

7.2 Marketing Automation Domain

The marketing agent ("Hanzo Growth") uses:

- **L3 configuration:** Research agent + copywriter agent + analytics agent + publisher agent.
- **Primary tools:** Web search, browser automation, data analysis, email/CMS publishing tools (via custom MCP extensions).
- **Default tier:** T_0 (Shared Gateway) for research and copywriting. T_2 (Operative Desktop) for browser-based campaign management.
- **Improvement signal:** Click-through rate, conversion rate, campaign ROI.

The marketing domain illustrates the Channel Integration layer's value: campaigns are triggered from Slack, executed autonomously, and results are reported back to Slack and the connected Linear project.

7.3 DevOps Operations Domain

The DevOps agent ("Hanzo Ops") uses:

- **L3 configuration:** Alert triage agent + diagnosis agent + remediation agent, connected to PagerDuty webhook via Layer 4.
- **Primary tools:** kubectl, Helm, Prometheus queries, log analysis, GitHub (deploy workflows).
- **Default tier:** T_3 (Cloud VM with direct cluster access).

- **Improvement signal:** Mean time to resolution (MTTR), false positive alert rate, successful remediation rate.

Over 90 days of production deployment, Hanzo Ops reduced mean MTTR from 47 minutes to 8.3 minutes for the class of incidents within its capability scope (database performance issues, deployment failures, certificate expirations, and capacity alerts).

7.4 Research Domain

The research agent ("Hanzo Research") uses:

- **L3 configuration:** Literature search agent + experimental design agent + data analysis agent + writing agent.
- **Primary tools:** ArXiv/PubMed search, PDF parsing, statistical analysis tools, Python/R execution, citation management.
- **Default tier:** T_3 (Cloud VM for compute-heavy analysis). Upgrades to T_4 (Local Desktop) when GPU acceleration is required.
- **Improvement signal:** Literature coverage completeness, statistical validity (verified by Reviewer agent), citation accuracy.

Table 3: Domain-Specific Performance Metrics

Domain	**Primary Metric**	**UAH Score**	**Default Tier**	**Improvement Loops**
Software Eng.	SWE-bench resolved rate	74.3%	T_1	4
Marketing Auto.	Campaign setup accuracy	81.2%	T_0/T_2	3
DevOps Ops.	Incident resolution rate	76.8%	T_3	4
Research	Literature coverage	88.4%	T_3/T_4	4
Mean		**80.2%**	—	—

Across all four domains, the infrastructure layers (L1, L2, L4, L5, L6, L7) are identical. Total additional code required to deploy a new domain is approximately 400–800 lines: the Layer 3 agent specification, any custom MCP tool extensions, and the domain-specific improvement signal definition.

8 Production Implementation

8.1 Infrastructure Overview

The Hanzo UAH is deployed on two Kubernetes clusters managed by DigitalOcean Kubernetes Service (DOKS):

- **hanzo-k8s** (24.199.76.156, 4 worker nodes, v1.34.1): Primary cluster hosting all Hanzo services including the UAH gateway, agent runtime, LLM proxy, IAM, and KMS.
- **lux-k8s** (24.144.69.101): Secondary cluster for Lux blockchain infrastructure (not directly relevant to the UAH but shares the observability stack).

8.2 Layer 1 Implementation: LLM Gateway

The LLM Access layer is implemented as the Hanzo LLM Gateway (`/llm`), a LiteLLM-based proxy extended with a custom routing engine. The gateway exposes an OpenAI-compatible API at `api.hanzo.ai` and `llm.hanzo.ai`, serving 100+ model providers through a single endpoint.

Key implementation details:

- **Routing engine:** A Go-based routing service that maintains a real-time capacity map, updated every 30 seconds via provider health checks. Routing decisions are made in < 1 ms.
- **Failover:** Exponential backoff with jitter on provider errors; automatic rerouting to a secondary provider within 200 ms.
- **Cost tracking:** Every API call is logged to PostgreSQL with model, provider, input tokens, output tokens, and calculated cost. Aggregate cost is queryable via Prometheus metrics.

8.3 Layer 2 Implementation: MCP Server

The Hanzo MCP Server (`/mcp`) implements the full Model Context Protocol specification and provides 260 tools (54 built-in plus 206 via plugin extensions). The server is available as an npm package (`@hanzo/mcp`) and as a Docker image for server deployment.

Architecture: a TypeScript Node.js server with tool handlers organized into category modules. Each tool handler is registered with the MCP runtime at startup, exposing a `tools/list` endpoint and handling `tools/call` RPC calls via JSON-RPC 2.0 over stdio or HTTP/SSE.

Sandbox isolation for shell execution tools uses `seccomp` profiles and Linux namespaces (via `runc`), providing filesystem, network, and process isolation without VM overhead. Measured overhead: < 5 ms per container spawn (using pre-forked process pool).

8.4 Layer 3 Implementation: Agent Runtime

The Agent Runtime is implemented as the Hanzo Agent SDK (`/agent`), a Python package providing the `Agent[TContext]` generic dataclass and the `AgentNetwork` orchestration engine. Workflow state is persisted in Temporal [Temporal, 2023] for durability and resumability: each agent step is a Temporal activity, and long-horizon tasks are Temporal workflows with up to 10-year retention.

Multi-agent routing is implemented as a tournament selection policy:

$$a^* = \underset{a \in \text{Candidates}(\tau)}{\arg\max} \; \text{CompatibilityScore}(a, \tau), \tag{12}$$

where CompatibilityScore is computed as the cosine similarity between the task embedding and the agent's skill embedding, both using `text-embedding-3-large` embeddings stored in Qdrant.

8.5 Layer 4 Implementation: Channel Integration

Channel adapters are implemented as independent microservices that normalize channel-specific message formats to the UAH canonical message schema. Each adapter maintains a persistent WebSocket connection to the NATS message bus [NATS, 2023], which routes messages to the appropriate agent instance.

The identity system (Hanzo IAM, deployed at `hanzo.id`) maps channel-specific identities to canonical users using the Casdoor platform [Casdoor, 2023] with custom identity federation extensions. All user data is encrypted at rest using Hanzo KMS (`kms.hanzo.ai`, based on Infisical [Infisical, 2023]).

8.6 Layer 5 Implementation: Compute Tiers

- **Tier 0:** Handled directly by the UAH gateway process; no separate provisioning.
- **Tier 1:** Docker containers on the hanzo-k8s cluster, orchestrated by a custom pod manager with a pool of 50 pre-warmed containers. Container images are based on `ubuntu:22.04` with standard development tools pre-installed.
- **Tier 2:** Operative Desktop pods with Xvfb, VNC server, and Chromium. Managed by the Hanzo Operative framework (`/operative`). VNC sessions are proxied via WebSocket to the agent's browser automation tools.
- **Tier 3:** DigitalOcean Droplets provisioned via the DigitalOcean API (4 vCPU / 8 GB baseline; scalable to 32 vCPU / 64 GB). Provisioned in ≤ 90 seconds from the UAH's pre-authenticated API client.
- **Tier 4:** Local machines register with the UAH gateway via a lightweight daemon (`hanzo-local-agent`), which establishes a Tailscale [Tailscale, 2023] encrypted tunnel and registers the machine's capabilities in the tier registry.

8.7 Layer 6 Implementation: Self-Improvement

The Self-Improvement layer is implemented as a set of periodic Temporal workflows:

- **Session summarizer** (runs on session close): Extracts key patterns from the session trace and updates the agent's episodic memory in Qdrant.
- **Hourly analyzer**: Aggregates the past hour's traces, identifies the top-10 failure modes by frequency, and generates candidate policy patches via an LLM introspection call.
- **Daily optimizer**: Evaluates candidate patches against a held-out validation set, selects improvements with Δsuccess $\geq 1\%$ and deploys them via a GitOps workflow.
- **Weekly fine-tuner**: Prepares a fine-tuning dataset from the week's successful traces and submits a fine-tuning job for domain-specific specialist models.

8.8 Layer 7 Implementation: Observability

All layers emit spans in OpenTelemetry format to an OpenTelemetry Collector deployed in the cluster. The collector fans out to:

- **Tempo** (Grafana): Distributed trace storage, queried via the Grafana trace explorer.
- **Loki** (Grafana): Structured log aggregation with LogQL query support.
- **Prometheus**: Metric scraping from all services; dashboards in Grafana.
- **AlertManager**: PagerDuty integration for critical alerts (agent failures, cost budget overruns, tier provisioning failures).

Every trace includes: `trace_id` (UUID), `session_id`, `user_id`, `agent_id`, `tier`, `model`, `provider`, `input_tokens`, `output_tokens`, `tool_calls` (array), `success` (bool), `cost_usd`, and `duration_ms`.

8.9 Total System Metrics

As of February 2026, the production UAH system processes:

- ∼12,000 agent sessions per day across all domains.
- ∼840,000 LLM API calls per day through the gateway.
- ∼3.2 million tool invocations per day.
- Peak Tier 1 concurrency: 180 active containers.
- Total observability volume: ∼4.8 GB compressed spans per day.
- P99 end-to-end session latency (Tier 0): 3.8 seconds.
- P99 tier upgrade latency (Tier 0→1, pre-warmed): 1.3 seconds.

9 Discussion and Future Work

9.1 Dynamic Mid-Session Tier Migration

The current tier transition model upgrades but does not freely migrate between arbitrary tiers mid-session. A session on Tier 3 cannot downgrade to Tier 2 without risking state loss if the running process has no safe checkpoint. Future work will develop a **live migration protocol** analogous to VM live migration [Clark et al., 2005]: the agent's execution state (open files, running processes, active connections) is serialized incrementally while the target environment is prepared, minimizing migration time and eliminating visible downtime.

9.2 Multiplayer Agent Collaboration

The current architecture supports multi-agent execution within a single session but does not address multiple human users collaborating on the same agent session in real time. We are developing a **collaborative session model** inspired by CRDTs [Shapiro et al., 2011]: each participant's instructions are modeled as operations on a shared session document, with conflict resolution handled by the orchestrator agent. This enables scenarios such as a developer and a product manager jointly directing an agent to implement a feature, with each able to inspect the agent's reasoning and interject corrections.

9.3 Autonomous Operation with Configurable Trust Boundaries

The current system requires human approval for certain actions (large cost expenditures, irreversible system changes, external communications). Future work will introduce a **trust boundary framework** that allows organizations to configure the set of actions an agent may take autonomously vs. those that require human approval, with cryptographically signed audit logs for all autonomous actions. This is related to AI safety work on corrigibility [Soares et al., 2015] and controllability, but tailored to the practical constraints of production deployment.

9.4 Federated Self-Improvement

The current self-improvement loops operate on a single organization's data. The Hanzo DSO (Decentralized Semantic Optimization) protocol [Hanzo AI, 2024a] defines a privacy-preserving mechanism for sharing improvements across organizations using zero-knowledge proofs and federated aggregation. Integration of DSO with the UAH's Layer 6 would allow the harness to improve from collective experience while preserving per-organization data isolation—analogous to federated learning [McMahan et al., 2017] but operating on structured semantic priors rather than gradient updates.

9.5 Limitations

The UAH has several notable limitations that bound its applicability:

1. **Task scope:** The harness performs best on tasks with measurable outcomes (code tests, metric targets). Open-ended creative tasks lack the feedback signal required for the self-improvement layer.

2. **Latency:** Tier upgrades and agent handoffs introduce latency overhead (1–90 seconds) that is unacceptable for real-time interactive applications. Tier 0 (sub-second response) must be used for latency-critical workflows.

3. **Cost predictability:** Because the harness dynamically selects compute tiers, costs per session can vary by up to 100× depending on task complexity. Budget caps and cost estimation help but do not eliminate cost uncertainty.

4. **Model dependence:** Despite routing across 100+ providers, the overall system performance is bounded by the capability of the underlying frontier models. Architectural improvements cannot substitute for model capability on tasks requiring reasoning beyond model capacity.

10 Conclusion

We have presented the Unified Agent Harness (UAH), a vertically integrated seven-layer architecture for autonomous AI agents. The central argument of this paper is both empirical and architectural: the harness, not the model, is the product.

The empirical case is demonstrated by the competitive analysis: the UAH achieves 74.3% on SWE-bench Verified, 70.2% better absolute performance than the next best multi-domain competitor, while maintaining competitive per-task cost. The performance advantage derives not from a superior underlying model but from cross-layer optimization enabled by vertical integration.

The architectural case is established by Theorem 1: any system that separately optimizes individual layers will yield strictly suboptimal end-to-end performance compared to joint optimization over all layers—under the empirically verified condition that layer performances are non-separable. The seven cross-layer effects we document empirically confirm this non-separability in production.

The broader implication is organizational as much as technical. As frontier model capabilities converge (closing the gap between providers that independently optimize LLMs), the durable competitive advantage in AI-powered products will come from the infrastructure layer: routing, tooling, compute, observability, and self-improvement. The AWS analogy holds: just as cloud infrastructure became the durable business layer above a commoditizing hardware market, the agent harness will become the durable business layer above a commoditizing model market.

The UAH defines what a complete agent harness must contain. We release the architecture specification, the formal layer definitions, and the open-source components (@hanzo/mcp, hanzo-agent, hanzo-operative) to the community, and invite the field to build upon, critique, and extend this foundation.

A Complete Layer Interface Specifications

A.1 ToolCallRequest Schema

```
ToolCallRequest {
  call_id:     UUID
  tool_name:   string
  parameters:  Map<string, JSON>
  context:     ExecutionContext
  timeout_ms:  uint32           // default: 30000
  idempotency: IdempotencyClass // SAFE | IDEMPOTENT | UNSAFE
}

ExecutionContext {
  session_id:  UUID
  agent_id:    string
  user_id:     UUID
  tier:        TierLevel        // T0..T4
  trace_id:    UUID
  capabilities: CapabilitySet
}
```

A.2 TierTransition Protocol

```
TierTransitionRequest {
  session_id:    UUID
  from_tier:     TierLevel
  to_tier:       TierLevel
  reason:        TransitionReason
  context_bundle: bytes          // CBOR-encoded, AES-256-GCM
  budget_usd:    float64
}

TierTransitionResponse {
  success:       bool
  environment:   ExecutionEnvironment
  migration_ms:  uint32
  estimated_cost_usd: float64
}
```

B SWE-bench Evaluation Setup

Evaluation followed the SWE-bench Verified [Jimenez et al., 2024] protocol:

- **Dataset:** SWE-bench Verified, 500 GitHub issues from 12 Python repositories.
- **Evaluation server:** Isolated Docker containers with the exact repository versions specified in each issue.
- **Success criterion:** All tests in the test suite pass after applying the agent's patch.
- **Compute:** Tier 1 (Terminal Pod) for patch generation; Tier 0 for issue analysis.
- **Model:** Zen-Code-480B (internal) via the UAH LLM gateway, with GPT-4o as a fallback for tool-use steps.
- **Maximum steps:** 50 agent steps per issue.
- **Time limit:** 20 minutes per issue.
- **Runs:** 3 independent runs; results are mean ± std. Standard deviation: ±1.2%.

C Tier Complexity Thresholds

The complexity thresholds θ_j used in Algorithm 1 were tuned on a 2,000-task calibration set:

Table 4: Tier Complexity Thresholds and Typical Task Types

Tier	$\boldsymbol{\theta_j}$	Typical Task Type	Example
T_0	$[0.0, 0.3)$	Q&A, summarization, translation	"What is RAG?"
T_1	$[0.3, 0.55)$	Code gen, data analysis, scripting	"Fix this bug"
T_2	$[0.55, 0.70)$	Web automation, GUI workflows	"Book a flight"
T_3	$[0.70, 0.88)$	System admin, multi-service tasks	"Deploy staging env"
T_4	$[0.88, 1.0]$	Hardware access, GPU compute	"Train classifier"

References

Aider. Aider: AI pair programming in your terminal. https://aider.chat, 2024.

Anthropic. The Claude 3 model family: Opus, Sonnet, Haiku. Technical report, Anthropic, 2024.

Anthropic. Introducing computer use, a new Claude 3.5 Sonnet, and Claude 3.5 Haiku. https://www.anthropic.com/news/3-5-models-and-computer-use, 2024.

AutoGPT Team. AutoGPT: An autonomous GPT-4 experiment. https://github.com/Significant-Gravitas/AutoGPT, 2023.

Brown, T., Mann, B., Ryder, N., Subbiah, M., et al. Language models are few-shot learners. *NeurIPS*, 33:1877–1901, 2020.

Casbin. Casdoor: An open-source identity and access management platform. https://casdoor.org, 2023.

Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. *NSDI*, 2:273–286, 2005.

Cognition Labs. Introducing Devin, the first AI software engineer. https://www.cognition-labs.com/blog, 2024.

Anysphere Inc. Cursor: The AI-first code editor. https://cursor.com, 2024.

Hanzo AI Research. Decentralized semantic optimization: Collaborative knowledge accumulation for AI agents. Technical report, Hanzo AI Inc, 2024.

Hanzo AI Research. Operative: A computer use framework for autonomous AI agents. Technical report, Hanzo AI Inc, 2024.

Hanzo AI Research. Hanzo agent SDK: Building AI agents with orchestration, tool use, and memory systems. Technical report, Hanzo AI Inc, 2024.

Infisical Inc. Infisical: Open-source secret management platform. https://infisical.com, 2023.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. SWE-bench: Can language models resolve real-world GitHub issues? *ICLR*, 2024.

Chase, H. LangChain: Building applications with LLMs through composability. https://github.com/langchain-ai/langchain, 2022.

McMahan, B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B. A. Communication-efficient learning of deep networks from decentralized data. *AISTATS*, pp. 1273–1282, 2017.

Anthropic. Model Context Protocol: An open standard for connecting AI models to tools and data. https://modelcontextprotocol.io, 2024.

Synadia Communications. NATS: Cloud native messaging. https://nats.io, 2023.

OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

OpenAI. Introducing Operator. https://openai.com/blog/introducing-operator, 2025.

Cloud Native Computing Foundation. OpenTelemetry: High-quality, ubiquitous, and portable telemetry. https://opentelemetry.io, 2023.

Park, J. S., O'Brien, J., Cai, C. J., Morris, M. R., Liang, P., and Bernstein, M. S. Generative agents: Interactive simulacra of human behavior. *UIST*, pp. 1–22, 2023.

Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. *Symposium on Self-Stabilizing Systems*, pp. 386–400, 2011.

Soares, N., Fallenstein, B., Yudkowsky, E., and Armstrong, S. Corrigibility. *AAAI Workshop on AI and Ethics*, 2015.

Tailscale Inc. Tailscale: Zero config VPN. https://tailscale.com, 2023.

Temporal Technologies. Temporal: Open source durable execution system. https://temporal.io, 2023.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., and Wang, C. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 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. *NeurIPS*, 2024.