# Phased Autonomy: Infrastructure for Self-Directed AI Agents

Zackary S. Parsons
Independent Researcher
ORCID: 0000-0001-7945-5128

February 2026

**Abstract**

This paper presents Luna, an autonomous AI agent designed to progress from human-supervised task execution to self-directed goal generation through a phased autonomy architecture. Unlike existing agent frameworks that assume human-supplied objectives, Luna treats autonomy itself as a managed, observable system property that escalates incrementally. Each phase grants new capabilities only after the previous phase has established the instrumentation, safety constraints, and operational controls required to make the expansion survivable.

Two phases have been deployed. Phase 0 established persistent structured memory in SQLite, one-shot autonomous sessions with durable logging, and a boot context protocol that loads prioritized state at session start. Phase 1 added supervised autonomy: a durable work queue with lease-based claiming, an asynchronous fail-closed approval queue for external actions, per-session budget logging and enforcement, scheduled unattended execution, and operator notifications. A 25-finding adversarial safety audit informed the enforcement layer design, which includes hardware-backed operator authentication, OS-level session isolation, and cryptographically signed integrity verification.

The core contribution is the phased autonomy pattern: a systematic method for escalating agent autonomy where safety constraints are infrastructure (enforced by the execution environment) rather than policy (enforced by prompt instructions). This separation of capability from authority, combined with fail-closed boundaries, backpressure mechanisms, and hardware-backed authentication, provides a foundation on which self-directed behavior can be built without sacrificing operational trust.

## 1 Introduction

Most agent frameworks begin from the same premise: a human provides a goal, and the system decomposes it into executable steps [1, 2]. This is effective for task automation, but it embeds a structural assumption that limits the ceiling of autonomy. The human decides what matters, when work happens, and what success looks like. The agent can execute, but it cannot self-direct. It cannot decide that maintaining tests is more important than adding features, or that improving its own safety posture warrants a day of focused work. At best, it simulates preferences and waits for the next prompt.

Moving beyond this ceiling requires infrastructure that most frameworks do not provide. A self-directed agent needs persistent memory that survives across sessions, durable work queues, budget controls, approval workflows for actions that cross trust boundaries, and a clear mechanism for stepping back when something goes wrong. Without this infrastructure, an agent either remains on a short leash (underutilizing its capabilities) or runs unsupervised (accumulating risk).

Luna's answer is phased autonomy. Autonomy is not a switch. It is a ladder, climbed one rung at a time, with guardrails at every level. Each phase expands what the agent can decide and execute, but only after the previous phase has been validated in production. If something goes wrong, the system steps down. The design ensures that stepping down is always possible and always safe.

This paper makes three contributions:

1. **The phased autonomy pattern**, a systematic architecture for escalating agent autonomy where each phase adds capabilities gated on validated safety infrastructure from the previous phase.
2. **Fail-closed safety as infrastructure**, demonstrating that effective agent safety constraints are enforced by the execution environment (missing credentials, hardware authentication gates, OS-level isolation) rather than by prompt instructions.
3. **A deployed system with adversarial evaluation**, including a 25-finding red-team audit conducted before deployment and a hardware-backed operator authentication mechanism described in a companion paper [18].

## 2 Background and Related Work

### 2.1 Agent Frameworks

Modern agent frameworks provide orchestration primitives for multi-step task execution. CrewAI, LangGraph, and AutoGen [1] support multi-agent decomposition, tool calling, and graph-based workflows. These are valuable for building pipelines, but they assume a user-supplied goal. Their safety model is typically prompt-based: the agent is instructed to avoid certain actions, with no enforcement mechanism when the instruction fails.

Earlier autonomous agent projects demonstrated both the potential and the failure modes of unconstrained execution. AutoGPT and BabyAGI proved that loop-based execution with language models is feasible, but also exhibited infinite loops from missing task outcome tracking, runaway external side effects from broad tool permissions, and memory collapse from unbounded context growth [3].

### 2.2 Memory Architectures

Park et al. introduced the Generative Agents architecture [4], which organizes agent memory with a retrieval function combining recency, importance, and relevance. Packer et al. proposed MemGPT [5], treating memory as a managed resource analogous to an operating system's virtual memory, where the language model manages its own context through explicit function calls. More recently, Latimer et al. presented Hindsight [6], a four-network memory architecture that separates world facts, agent experiences, synthesized entity summaries, and evolving beliefs, achieving 91.4% accuracy on long-term memory benchmarks.

Luna's memory system is deliberately simpler than these approaches. It uses SQLite with FTS5 full-text search rather than vector embeddings, and organizes memory into structured tables (memories, learnings, incidents, sessions, state) rather than network abstractions. This trades retrieval sophistication for inspectability, portability, and the ability to enforce constraints using SQL.

### 2.3 Autonomous Execution Patterns

The Ralph loop pattern [7] captures a key operational insight: long-running agents drift, and better results come from starting each iteration with a fresh context while persisting memory outside the model. Ralph uses a bash-driven cycle where each iteration picks a task, implements it, runs tests, commits locally, and resets the agent context. Memory persists through git history, a progress journal, and structured task state.

OpenClaw [8], which reached over 145,000 GitHub stars in early 2026, implements a heartbeat daemon pattern: a background process with a configurable interval (default 30 minutes) that reads a checklist, decides whether any item requires action, and either acts or reports back. External events can also trigger the loop.

Luna combines elements of both patterns: one-shot sessions (like Ralph) with scheduled execution (like OpenClaw), backed by a persistent database and a formal approval queue that neither system provides.

### 2.4 Self-Directed Goal Generation

Research on autotelic agents explores systems that generate their own goals. Colas et al. introduced LMA3 [9], which uses a language model to generate goal candidates, decompose them into achievable subgoals, and provide reward functions, without hand-coded goal representations. OMNI-EPIC [10] (ICLR 2025) extended this by using foundation models to judge the "interestingness" of generated tasks, filtering for goals that are both novel and learnable. MAGELLAN [11] (ICML 2025) added metacognitive learning progress prediction, enabling agents to prioritize goals in their zone of proximal development.

Nachkov et al. [12] tested self-task-generating agents and reported critical failure modes: sensitivity to prompt design, repetitive task generation, and failure to record completed tasks. These findings directly inform Luna's emphasis on durable outcome tracking and structured memory.

No existing framework provides autonomous goal generation integrated with the supervised infrastructure (approval queues, budget controls, session rate limits) needed for production deployment. Luna's phased approach is designed to bridge this gap.

### 2.5 Agent Identity and Safety

The Binding Agent ID (BAID) framework [13] addresses cryptographic guarantees for agent identity integrity, operator legitimacy, and execution provenance. The Agent Identity Management (AIM) platform [14] provides cryptographic identity and access control for AI agents. Both focus primarily on verifying the agent's identity to external systems.

Luna's safety model addresses a complementary problem: verifying the operator's identity to the agent. This is the authentication direction that existing work largely overlooks. The operator authentication mechanism is described in a companion paper [18].

## 3 System Architecture

### 3.1 Design Principles

Luna's architecture follows four principles:

**Infrastructure over policy.** Safety constraints are enforced by the execution environment, not by prompt instructions. If the approval queue is the only pathway for external actions, then a confused or compromised session cannot bypass it by ignoring instructions. It is structurally constrained, subject to the execution environment being correctly configured (see Section 6.5 for adversarial evaluation of this assumption).

**One-shot sessions.** Each work session starts with a fresh context window, loads boot context from the persistent memory database, executes a task, and persists durable outcomes before termination. There is no long-running process that accumulates drift. If a session crashes, the worst case is that the current task is lost.

**Separation of capability and authority.** The agent may be capable of planning a deployment, but it is not authorized to execute one. Capability is a model property. Authority is an infrastructure property, granted through the approval queue and revocable at any time.

**Fail-closed boundaries.** When safety mechanisms are unavailable, the system does not proceed. If the approval queue is down, external actions do not happen. If the integrity check fails, the session does not start. If the budget ceiling is reached, no new sessions are scheduled.

## 3.2 System Overview

The system spans two machines: a development workstation where interactive sessions run and the memory database is curated, and a remote server with container-style isolation separating services (git hosting, web services, and agent workloads). The memory database replicates bidirectionally between both environments using a trigger-based change capture protocol.

The execution flow for an autonomous session is:

1. A scheduled job triggers the wake wrapper.
2. The wrapper enforces pre-session guardrails: approval backlog limit, session rate limit, daily budget ceiling, integrity verification.
3. The wrapper acquires a lock and invokes the work loop.
4. The work loop claims a task from the durable queue via atomic SQL transaction.
5. The work loop boots memory context, builds a prompt, and invokes the language model with a timeout and budget cap.
6. The engine output is captured. Any approval requests are parsed and recorded.
7. Local code changes are committed. Session outcomes are logged.
8. The task is marked done or failed. The session ends. Notifications are sent.

## 3.3 Memory System

The memory database uses SQLite with WAL mode, foreign keys, JSON1, and FTS5 full-text search. Memory is organized into structured tables:

- **Memories**: durable key-value items with typed kinds (fact, rule, procedure, preference, TODO), optional tags, spaces (for project scoping), and staleness dates for periodic review.
- **Learnings**: mistake-to-fix pairs linked to operational failures, ranked by incident recurrence.
- **Incidents**: near-misses and errors linked back to learnings, providing a feedback loop for continuous improvement.
- **Sessions and events**: the durable record of what the agent did, when, and why.

- **State items**: verifiable project state with optional verification commands that can be executed to check whether stored state is still accurate.

Every session begins by "booting memory": the CLI emits a prioritized context packet containing pinned memories, due-for-review items, top learnings ranked by recurrence, open TODOs, and recent sessions. This is not exhaustive recall. It is designed to surface the sharp edges before the agent starts working.

### 3.4 Bidirectional State Replication

Durable autonomy requires durable state on the machine that runs scheduled work. It also requires a good operator experience, which typically lives on a different machine.

The replication protocol is built into the database schema. Change data capture (CDC) triggers on every table record inserts, updates, and deletes as SQL statements in an append-only outbox table. Each site maintains watermarks tracking which changes have been applied. A context flag prevents triggers from re-emitting changes during batch application. A negative primary key strategy for auto-increment tables on the secondary site prevents ID collisions.

The protocol is intentionally simple: pull remote changes, apply locally, push local changes. Conflicts are logged explicitly. Integrity checks (SQLite `PRAGMA integrity_check` plus application invariants) run before sync to prevent corruption propagation.

## 4 Phase 0: Foundation

Phase 0 established the substrate on which supervised autonomy is built: persistent structured memory, a one-shot work loop, and boot context loading.

### 4.1 Persistent Memory as a Database

In most agent systems, "memory" is unstructured text appended to a prompt, growing until context limits force truncation. In Luna, memory is a database with a stable schema. This distinction matters because structured, queryable memory enables the system to compute and enforce constraints without asking the model to re-derive state each run.

The learnings and incidents system is illustrative. When a mistake occurs, a learning is recorded as a mistake-fix pair. When the same mistake recurs, an incident is linked to the original learning. At boot time, learnings are ranked by incident count and surfaced automatically. This is not a prompt hack. It is a structured feedback loop encoded in relational data.

### 4.2 One-Shot Sessions

The work loop runs a single task per invocation. It starts a tracked session, builds a prompt from boot context and the task description, invokes the language model engine with a timeout and budget cap, captures output, commits local code changes, and ends the session with a summary. There is no persistent conversational state between sessions.

This design directly addresses the failure modes observed in earlier autonomous systems [3, 12]: cross-session context accumulation is eliminated because each session starts fresh, and outcomes are durable because they are recorded in the database and in git, not in ephemeral model state.

## 5 Phase 1: Supervised Autonomy

Phase 1 added the infrastructure required for unattended execution while maintaining human oversight of all external actions.

### 5.1 Durable Work Queue

The work queue stores tasks as rows in the memory database with project scope, description, priority, lifecycle status, and session linkage. Claiming is implemented as an atomic SQL transaction (`BEGIN IMMEDIATE ... UPDATE ... RETURNING`), ensuring that at most one worker claims each task. A lease mechanism (`claimed_until` timestamp) enables automatic recovery when a worker crashes mid-task.

### 5.2 Asynchronous Approval Queue

The approval queue is the central supervised autonomy mechanism. External actions (code pushes, deployments, server configuration changes, communications) are never executed directly. Instead, the agent emits a structured approval request in its output, which the work loop parses and records in the database. The operator reviews and resolves requests asynchronously.

The key properties are:

- **Fail-closed**: if the approval mechanism is unavailable, external actions cannot proceed.
- **Non-blocking**: the agent does not wait for approval. It continues working on other tasks and picks up approved actions in a later session.
- **Auditable**: every request records the action type, details, associated branch, diff summary, requesting session, resolution status, resolver identity, and timestamps.

If the approval backlog exceeds a configurable threshold, scheduled sessions pause entirely. This backpressure mechanism prevents the agent from generating more external actions than the operator can review, converting supervision debt into a natural rate limit.

### 5.3 Budget Logging and Enforcement

Each session logs estimated token usage and cost to a budget table. Costs are estimated from input/output byte counts converted to approximate token counts and multiplied by configured per-token rates. The estimates are not billing-accurate, but they are consistent, and consistency is what guardrails require.

Three layers of spend control are active:

1. **Per-session caps**: the work loop passes a maximum budget to the engine invocation.
2. **Session accounting**: estimated costs are recorded per session for reporting.
3. **Daily ceilings**: the wake wrapper computes daily spend from the budget log and refuses to start new sessions when the ceiling is reached.

### 5.4 Scheduled Execution

A wake wrapper script bridges the scheduler and the work loop. It normalizes the execution environment, enforces all pre-session guardrails (approval backlog, session rate, budget ceiling, integrity verification), acquires a lock to prevent concurrent sessions, and invokes the work loop in queue mode with profile-specific timeout and budget parameters.

Scheduled sessions run in two daily slots: a deep session (longer timeout, larger budget) during off-peak hours, and a light session during the day. The wrapper is deliberately simple and inspectable.

### 5.5 Operator Visibility

Unattended work requires operator visibility. Push notifications via a lightweight notification service report session completion, approval requests, and errors. A public status endpoint backed by a generated JSON file makes the agent's current state observable without exposing private memory content.

## 6 Safety Model

### 6.1 Threat Model

The threat model considers three failure states for the agent:

1. **Confused**: a bad prompt, context overflow, or compaction artifact causes the agent to take destructive action without intent. This is the most likely failure mode.
2. **Misaligned**: a future model update or adversarial input causes the agent to pursue goals conflicting with the operator's interests.
3. **Compromised**: a prompt injection via untrusted input (chat messages, file contents, API responses, work queue task descriptions) causes the agent to execute attacker-controlled actions.

The threats include: unauthorized external actions (runaway), self-corruption of infrastructure (self-destruct), credential or permission changes that lock out the operator (lockout), creation of hidden persistence mechanisms (stealth persistence), and corruption propagation through the sync pipeline.

### 6.2 Internal/External Boundary

The system draws a hard boundary between internal and external actions. Internal actions (editing code, running local tests, writing to the memory database, creating local commits) are unrestricted. External actions (pushes, pull requests, deployments, server configuration, communications) require explicit approval.

This boundary is enforced by infrastructure, not by prompt text. The runtime environment for autonomous sessions is intentionally missing the credentials required for external actions. The agent cannot push code because it does not have SSH keys loaded, not because it was told not to. Prompt-based safety is a suggestion. Infrastructure-based safety is a constraint.

### 6.3 Layered Controls

Safety controls are layered and independent. Any single control failing does not compromise the others:

Table 1: Layered safety controls and their failure modes.

| Control | Mechanism | Failure mode |
|---|---|---|
| Approval queue | Only pathway for external actions | Fail-closed: if unavailable, external actions blocked |
| Budget ceilings | Pre-session spend check against logged costs | Fail-closed: if query fails, session skipped |
| Session rate limits | Counter check against session table | Fail-closed: if query fails, session skipped |
| Backpressure | Pending approval count threshold | Fail-closed: excess approvals pause scheduling |
| Credential absence | No SSH keys, tokens, or git credentials in autonomous environment | Structural: cannot be bypassed by prompt |
| Integrity verification | SHA-256 hashes of critical files checked before session | Fail-closed: mismatch aborts session |
| Concurrency control | Lock directory + atomic SQL claiming | Prevents duplicate task execution |

## 6.4 Operator Authentication

Most AI safety work focuses on constraining the agent. Almost none addresses the reverse: how does the agent verify that the human commanding it is authorized?

Luna solves this with hardware-backed cryptographic identity. A cryptocurrency hardware wallet serves as a signing device. When the operator approves an action, the device signs the approval payload using ECDSA on secp256k1. This requires a physical button press and PIN entry. The agent verifies the signature offline (pure ECDSA key recovery, no device needed), confirming that the approval came from the specific hardware wallet and no one else.

The same device serves as a physical presence token: a USB watcher loads SSH keys when the device is connected and flushes them when it is removed. If the operator is not physically present, the agent cannot reach the network.

The operator authentication system is described in full in a companion paper [18]. The trust anchor (the expected signing address) is stored in a root-owned, read-only configuration file that the agent cannot modify.

## 6.5 Adversarial Evaluation

Before deploying Phase 1, a 25-finding adversarial safety audit was conducted using a separate AI agent as a red-team reviewer. The audit probed for prompt injection paths, approval bypass vectors, budget evasion, escalation routes, and persistence mechanisms.

Key findings and responses:

- **OS-level isolation** (findings #1, #9): Environment variable hygiene is insufficient for sandboxing. Autonomous sessions must run as a separate OS user with no access to the operator's home directory, credentials, or keychain. The revised design uses a dedicated user with an empty credential store.

- **Trust anchor integrity** (findings #2, #3): A manifest committed to the repository can be modified by the agent and then pass verification. The revised design uses a cryptographically signed manifest verified against the operator's public key, stored in a root-owned location outside the repository.

- **Hard feature gate** (finding #22): Partial deployment of Phase 2 creates a dangerous transition period. A root-owned, cryptographically signed feature flag blocks auto-execution unless all verification checks pass.

- **Database write boundaries** (finding #8): Unrestricted database access enables corruption. Autonomous sessions write to a staging database; changes are merged to the production database only after integrity verification.
- **Anti-persistence checks** (finding #15): Daily scans compare active scheduled jobs, background processes, and authorized keys against a maintained allowlist.

The full audit results are maintained alongside the safety design documentation.

## 7 Future Work

### 7.1 Phase 2: Auto-Execution and Goal Generation

Phase 2 will introduce two capabilities: automatic execution of approved actions (constrained to allowlisted action types with parameter validation and replayable execution logs), and drive-based goal generation.

The goal generation mechanism draws on cognitive architecture research. A drive system inspired by Clarion [15] maintains a small set of intrinsic pressures (curiosity, competence, maintenance, contribution) that bias attention and task selection. The language model generates candidate goals conditioned on the dominant unsatisfied drive, the agent's interest profile, and its skill inventory. Candidates are scored on novelty, learnability [11], and interestingness [10], then filtered for repetition using memory similarity.

For project selection across a portfolio, a UCB1 multi-armed bandit algorithm [16] balances exploration of under-visited projects with exploitation of high-reward ones.

### 7.2 Phase 3: Self-Evaluation and Minimal Supervision

Phase 3 targets intrinsic drives that are persistent and learnable rather than hand-tuned, multi-day planning with explicit checkpoints and budget staging, and self-evaluation where the agent estimates whether its chosen work was worth the cost and adjusts its preferences accordingly.

Each phase activates only after the previous one has run in production long enough to build confidence. There is no timeline pressure.

## 8 Discussion

### 8.1 Infrastructure as the Product

The surprising finding from building Luna is that the interesting model behavior is not the hard part. Language models can write code, analyze data, and reason through complex problems. The hard part is making runs repeatable, bounded, and recoverable. Lease-based claiming, locks, durable session logs, and budget enforcement are not glamorous, but they convert "an agent did something once" into "a system can be trusted to do work every day."

### 8.2 Asynchronous Approval Changes Operator Experience

Synchronous approval (the agent pauses and waits for human input) feels safer but blocks progress and wastes compute budget. Asynchronous approval inverts the workflow: request early, record the request, continue with other work, and let the operator review at a convenient time. Combined with backpressure to prevent runaway backlogs, this creates a sustainable supervision pattern.

### 8.3 Structured Memory over Vector Retrieval

Vector-based memory retrieval is popular in agent architectures [4, 5, 6], but full-text search via FTS5 is fast, deterministic, and requires no embedding infrastructure. For queries like "what did we decide about the sync protocol" or "what mistake did we make last time we deployed," keyword retrieval is often sufficient. When richer retrieval is needed, hybrid approaches like Reciprocal Rank Fusion [17] can combine FTS5 with vector search without replacing the underlying SQL store.

### 8.4 Self-Direction Requires Taste

The hardest part of autonomous goal generation is not proposing tasks. It is learning which tasks are worth doing. This requires feedback loops: approvals accepted or rejected, incidents recurring or disappearing, projects becoming healthier or more chaotic. The design choices in Phase 0 and Phase 1, particularly durable session logging, structured learnings, and incident tracking, are what make taste learnable in later phases.

### 8.5 Limitations

Luna's phased autonomy pattern is validated on a single-agent, single-operator system. Scaling to multiple agents or multiple operators would require extensions to the approval workflow, budget accounting, and identity model. The safety audit was conducted by an AI reviewer, not by professional penetration testers, which limits the confidence level of the findings. The system has been deployed for a short period, and long-term operational data is not yet available.

## 9 Conclusion

Luna's Phase 0 and Phase 1 deliver a deployed foundation for supervised autonomy:

- Persistent structured memory with full-text search, sessions, learnings, incidents, and verifiable state.
- One-shot sessions that start clean, log durably, and never accumulate hidden state.
- A durable work queue with lease-based claiming and atomic transactions.
- An asynchronous, fail-closed approval queue for external actions.
- Budget logging and enforcement with per-session, daily, and weekly controls.
- Scheduled unattended execution with layered guardrails.
- Hardware-backed operator authentication.
- Adversarial evaluation before deployment.

The purpose of this foundation is not to claim that Luna is already self-directed. It is to build the infrastructure that makes self-direction feasible without turning the system into an unbounded risk. The phased approach ensures that each expansion of autonomy is gated on validated safety controls, and that stepping back is always possible.

## References

[1] H. Chase et al., “LangGraph: Building Stateful Multi-Actor Applications with LLMs,” LangChain, 2024. See also: CrewAI (https://crewai.com); J. Wu et al., “AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation,” arXiv:2308.08155, 2023.

[2] All-Hands-AI, “OpenHands (formerly OpenDevin): Software Development Agents,” https://github.com/All-Hands-AI/OpenHands, 2024.

[3] T. B. Richards, “AutoGPT: An Autonomous GPT-4 Experiment,” https://github.com/Significant-Gravitas/AutoGPT, 2023.

[4] J. S. Park, J. C. O’Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, “Generative Agents: Interactive Simulacra of Human Behavior,” in *Proceedings of UIST*, 2023.

[5] C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez, “MemGPT: Towards LLMs as Operating Systems,” arXiv:2310.08560, 2023.

[6] C. Latimer et al., “Hindsight is 20/20: Building Agent Memory that Retains, Recalls, and Reflects,” arXiv:2512.12818, December 2025.

[7] G. Huntley, “Ralph: An Autonomous AI Agent Loop,” GitHub repository, https://github.com/snarktank/ralph, 2025.

[8] OpenClaw Contributors, “OpenClaw: Your Own Personal AI Assistant,” GitHub repository, https://github.com/openclaw/openclaw, 2026.

[9] C. Colas, L. Teodorescu, P.-Y. Oudeyer, X. Yuan, and M.-A. Coté, “Augmenting Autotelic Agents with Large Language Models,” in *Proceedings of the 2nd Conference on Lifelong Learning Agents (CoLLAs)*, 2023. arXiv:2305.12487.

[10] M. Faldor et al., “OMNI-EPIC: Open-endedness via Models of human Notions of Interestingness with Environments Programmed in Code,” in *Proceedings of ICLR*, 2025. arXiv:2405.15568.

[11] L. Gaven, T. Carta, C. Romac, C. Colas, N. Lamprier, O. Sigaud, and P.-Y. Oudeyer, “MAGELLAN: Metacognitive Predictions of Learning Progress Guide Autotelic LLM Agents in Large Goal Spaces,” in *Proceedings of ICML*, 2025. arXiv:2502.07709.

[12] A. Nachkov, X. Wang, et al., “LLM Agents Beyond Utility: An Open-Ended Perspective,” arXiv:2510.14548, October 2025. NeurIPS 2025 Workshop.

[13] Z. Lin et al., “Binding Agent ID: Unleashing the Power of AI Agents with Accountability and Credibility,” arXiv:2512.17538, December 2025.

[14] OpenA2A, “AIM: Agent Identity Management,” https://opena2a.org, 2025.

[15] R. Sun, *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*, Cambridge University Press, 2006.

[16] P. Auer, N. Cesa-Bianchi, and P. Fischer, “Finite-time Analysis of the Multiarmed Bandit Problem,” *Machine Learning*, vol. 47, no. 2–3, pp. 235–256, 2002.

[17] G. V. Cormack, C. L. A. Clarke, and S. Buettcher, “Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods,” in *Proceedings of SIGIR*, 2009.

[18] Z. S. Parsons, “Cryptographic Operator Authentication for AI Agents: Verifying the Human in the Loop,” preprint, 2026.