Indra: An Operating System for Large-Scale AI Agents

Mehul Srivastava
Five Labs
mehul@fivelabs.co

Whitepaper. July 2025.

Abstract

We present *Indra*, a modular operating system for orchestrating AI agents. Inspired by the collective intelligence of honeybee swarms, Indra pairs a planning *Mother-LLM* (the "Queen") with specialist *Worker* agents that communicate through JSON-RPC. The design brings replay, auditability, and cost-aware scheduling, which existing "agent" frameworks leave to the application developer. We present the architecture, an execution model in which every agent call is a metered state transition, the security primitives the design calls for, and a credit system that distills compute and API charges into a single fungible unit. We outline an evaluation plan across tool invocation, goal consistency, and structured-output adherence, building on public tool-use benchmarks; no evaluation results are reported. A reference implementation covering the orchestration loop is public under the Five Labs Community License, and Section 7 states exactly which parts of the design it implements.

1 Introduction

Generative AI has unlocked immense automation potential, but most "AI assistants" remain single-turn copilots that lack memory, fault tolerance, and the ability to execute multistep workflows. Knowledge workers still report spending close to 60 % of their day on coordination rather than skilled work (58 % in Asana's 2023 global index [1]). Recent surveys on LLM-based multi-agent systems highlight both rapid progress and open challenges [6, 7]. We argue that the missing layer is *orchestration*: a framework that routes context, tools, and credentials to the right model at the right time, and can account for what that cost.

Drawing on eusocial insects, specifically the task division and adaptive resilience of honeybee colonies [9], we propose **Indra**, a swarm-based operating system (OS) for distributed cognition among specialised AI agents. Indra treats every agent call as a transaction, supports deterministic replay, and uses an *Agent-to-Agent* (A2A) gateway for typed JSON-RPC communication. Figure 1 gives the overview.

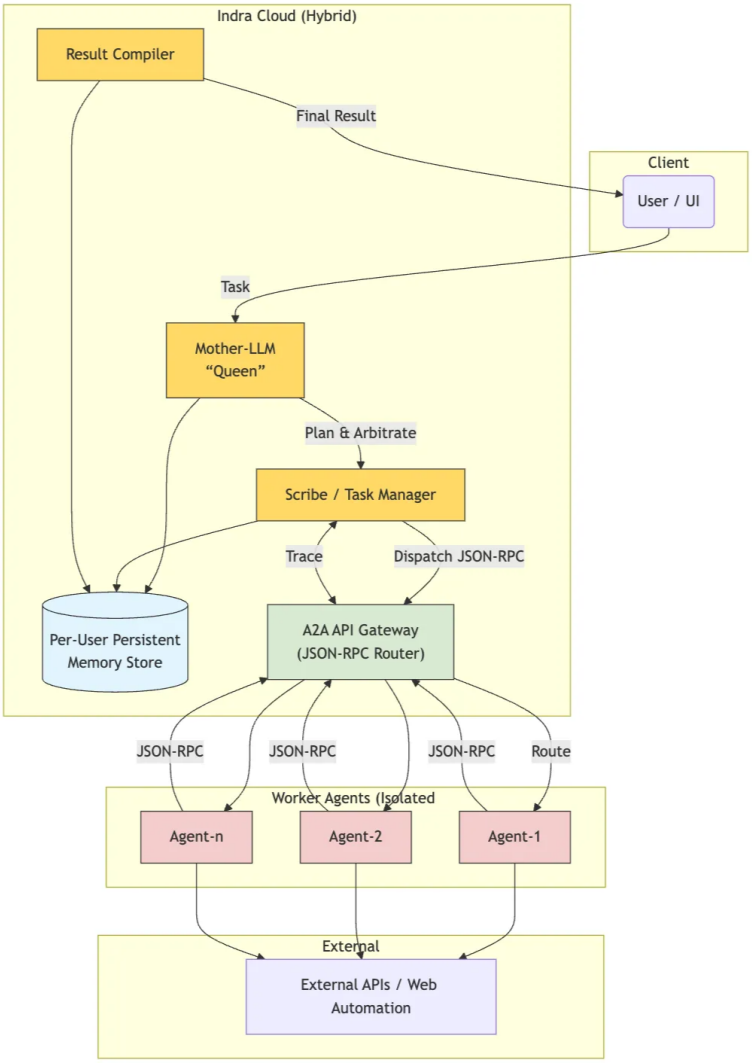

Figure 1: High-level architecture of Indra. The Queen plans, the Scribe allocates, Worker agents execute, and the Compiler returns the consolidated result to the client.

2 System Architecture

This section describes the target design. Where the reference implementation departs from it, Section 7 says so; the present tense below describes the design, not a deployed system.

Figure 1 shows the layered design, intended to run within *Indra Cloud*, a hybrid cluster of GPU and CPU nodes. Each tenant receives an isolated namespace with role-based access control (RBAC), which allows an enterprise to pin workloads to local nodes while shedding to public GPUs when policy permits.

2.1 Mother-LLM ("Queen")

The Queen is a managed service that wraps a frontier-grade language model (at the time of writing, Claude Opus 4 or Gemini 2.5 Pro). Plans are expressed in a mini-DSL we call `BeeScript`, a JSON serialisation of hierarchical tasks:

Listing 1: Excerpt of a BeeScript plan generated by the Queen

```
{
  "goal": "Generate Q2 marketing report",
  "budget_credits": 1200,
  "subtasks": [
    { "id": "gather.metrics", "agent": "fetcher", "params": {"period": "2025-Q2"}
       },
    { "id": "draft.report",   "agent": "writer",  "params": {"style": "pptx"},
      "after": ["gather.metrics"] },
    { "id": "qa",             "agent": "reviewer", "params": {},
      "after": ["draft.report"] }
  ]
}
```

The Queen's prompt template is version-controlled; a minimal example with placeholders:

Listing 2: Queen prompt template example

```
SYSTEM: You are the Queen planner in Indra OS. Produce a valid BeeScript JSON plan
    .
USER_TASK: {{user_intent}}
TOOLS_AVAILABLE: {{tool_catalogue}}
CONSTRAINTS: budget <= {{budget}} credits; depth <= 3.
```

Plan objects pass through a schema validator before execution. The validator checks task identifiers, dependency references, cycles, and the credit budget; malformed plans are patched where the fix is mechanical and rejected otherwise.

2.2 Scribe / Task Manager

The Scribe is designed as a Rust service on the Actix async runtime. It converts the BeeScript DAG into a runtime graph, tracks agents over WebSocket, and launches tasks by posting `/dispatch` calls to the A2A Gateway. Circuit-breakers are encoded as Prometheus alert rules; exceeding a credit allotment raises an event that pauses the offending agent and triggers the Queen to re-plan.

2.3 Agent-to-Agent Gateway

The Gateway is designed as a FastAPI service that routes JSON-RPC 2.0 messages over HTTP/2. It enforces:

1. **Schema validation**: requests must conform to the autogenerated Pydantic models.

2. **Rate limiting**: a leaky-bucket algorithm keyed by `agent_id` and `user_id`.

3. **Trace propagation**: each hop appends a SHA-256 hash of the payload to the `trace_chain` array.

Agent payloads travel over an authenticated duplex stream conforming to the *Indra Envelope*, whose key properties are:

- idempotent replays via a session nonce;
- pluggable transport (gRPC or WebSocket);
- a deterministic hash of the task graph for audit.

The reference implementation[1] exposes the Queen, Router, and Compiler as Python classes (`from indra import Queen, Router, Compiler`); workers subclass `BaseWorker` and register with a `@register_worker` decorator. The gateway itself is not implemented (Section 7).

2.4 Worker Agents

Workers are designed to run as OCI containers (`distroless-python` base) with a side-car that handles RPC marshaling. Two reference agent types are specified:

- **SLM agents**: lightweight models accessed via vLLM (e.g., `Qwen 2.5`).
- **Deterministic agents**: pure code with unit tests shipped alongside (e.g., a Pandas report generator).

Developers define an agent with a prompt file and a handler, e.g.:

Listing 3: Skeleton of a deterministic worker agent, in the design's decorator form

```python
from indra import agent

@agent(name="fetcher", cost=5)
def get_metrics(campaign_id: str, period: str):
    # code to query internal warehouse
    return metrics_df.to_dict()
```

2.5 Result Compiler

The Compiler assembles partial outputs via a Map-Reduce pattern. In the design, conflicts are resolved by an LLM-based ranker that scores the relevance of competing snippets, and final renderers include Markdown, HTML, PDF, and PPTX.

[1] https://github.com/Five-Research/indra-mvp

2.6 Implementation Footprint

The target footprint for a minimal deployment (Queen, Gateway, Scribe, three Worker agents) is a 16-vCPU, 64-GB RAM node plus two A100 40GB GPUs. Cold-start latency is unmeasured on the reference implementation, which runs in a single Python process.

3 Execution Model

Indra introduces the *Indra Virtual Agent Machine* (IVAM). A global state σ encompasses a vector memory $\mathcal{M}$, a credit balance $C \in \mathbb{N}$, and an append-only trace $\mathcal{T}$. A transaction τ is an RPC call with parameters p and fee f. The state transition function Γ is:

$$\Gamma(\sigma, \tau) \to \sigma' : \left[C' = C - f,\ \mathcal{T}' = \mathcal{T} \| \tau,\ \mathcal{M}' = \mathcal{R}(\mathcal{M}, p)\right] \tag{1}$$

where $\mathcal{R}$ denotes the optional read/write retrieval to memory. Execution halts when $C \leq 0$ or an agent returns FINISHED.

Extended state: leases and gas. In addition to $(\mathcal{M}, C, \mathcal{T})$ we maintain a *lease table* $\mathcal{L} = \{(\mathrm{id}, t_{\mathrm{exp}})\}$ that tracks the time-to-live for long-running workers. When the wall clock $\tau_{\mathrm{now}} \geq t_{\mathrm{exp}}$, the Scribe force-cancels the worker and triggers Γ with a synthetic TIMEOUT transaction that burns the remaining prepaid fee. We further decompose the credit balance as

$$C = G_{\mathrm{gas}} + G_{\mathrm{storage}},$$

mirroring EVM-style gas for CPU/GPU time versus persistent vector slots.

Composite (nested) transactions. A single user call often spawns an internal tree of RPCs. We model this with a *composite transaction* $\tilde{\tau} = \langle \tau_1, \ldots, \tau_k \rangle$ whose execution is atomic with respect to the external state:

$$\Gamma(\sigma, \tilde{\tau}) = \Gamma\big(\ldots \Gamma(\Gamma(\sigma, \tau_1), \tau_2) \ldots, \tau_k\big).$$

If any child returns ABORT, the interpreter returns to the parent snapshot σ and refunds the unused G_{gas}.

Parallel scheduling. IVAM partitions $\tilde{\tau}$ into an antichain $\mathcal{A} = \{\tau_i : \nexists\, \tau_j \prec \tau_i\}$ and dispatches those independent leaves to a work-stealing executor (similar to Ray's actor pools). Let λ be the per-GPU throughput; the expected latency is bounded above by $\mathbb{E}[T] \leq \big(|\mathcal{A}|/\lambda\big) + \delta_{\mathrm{sync}}$, where δ_{sync} is the barrier cost of merging the result.

Variable retry budget. Each τ carries a retry counter $r_{\max}$. On non-deterministic failure (network flake, LLM 5xx) we set $r \leftarrow r - 1$ and resubmit with exponential jitter $J \sim \mathrm{Uniform}(0, 2^k)$ s

where $k = r_{\max} - r$. The retry gas fee is $f_{\text{retry}} = \eta \cdot 2^k$ with damping factor $\eta \in (0, 1)$ to discourage runaway loops.

Deterministic replay. Every transition emits a keccak-256 hash $h_i = \mathsf{K256}(\sigma_i, \tau_i)$. The chain $\mathcal{H} = \langle h_1, \ldots, h_n \rangle$ can be anchored to an external data-availability layer (e.g., EigenLayer DA) so that an external verifier can replay the trace and reach the exact σ_n. Anchoring is optional in the design and absent from the reference implementation.

Memory eviction policy. Vector memory $\mathcal{M}$ is specified as an append-only Faiss index plus a log-structured merge tree (LSM) for scalar keys. When $|\mathcal{M}|$ exceeds a tenant-set threshold Θ, the eviction operator $\mathcal{E}$ chooses the entry with the lowest *usage-weighted value*

$$v(x) = \alpha \cdot \text{access\_freq}(x) + \beta \cdot \text{score}(x)$$

and swaps it to cold object storage, freeing live RAM without breaking referential integrity.

4 A2A Protocol

Google open-sourced the *Agent2Agent (A2A)* specification in April 2025 [2]: an HTTPS, JSON-RPC 2.0, and Server-Sent Events based protocol that standardises how autonomous agents discover each other, publish capabilities, and negotiate task delegation. Indra's design is a superset that stays wire-compatible with the reference specification while adding deterministic replay and credit metering.

Indra-specific extensions

- **Persistent memory token**: each RPC carries a `mem_id` that points to an IVAM vector-store slot, so a worker can recall long-term user context without extra lookups.
- **Goal context**: the header `x-indra-goal` embeds a one-line summary of the objective so that every worker knows *why* it is running a subtask.
- **Deterministic replay**: the header `x-indra-replay` carries a hash of the task graph, allowing exact re-execution for audits and debugging.
- **Credit metering**: the header `x-indra-credits` sets the maximum gas the callee may burn, bounded by the caller's balance.
- **Authentication and trust**: agents perform mutual authentication via OAuth 2.0 access tokens issued as JWTs.

4.1 Agent Cards

Each agent exposes a signed **Agent Card**, a compact JSON document served from `/.well-known/agent.json` as in the A2A specification. Cards declare metadata, capability verbs, endpoint URLs, and required scopes so that peers can perform capability matching. The listing below is illustrative; field names follow the spirit of the specification rather than a particular schema version.

Listing 4: Illustrative Agent Card for an Indra worker

```
{
  "id": "agent://indra/fetcher",
  "name": "Metrics Fetcher",
  "description": "Retrieves marketing metrics from the warehouse",
  "auth": {
    "type": "oauth2",
    "token_endpoint": "https://auth.indra.cloud/token",
    "scopes": ["metrics.read"]
  },
  "endpoints": {
    "rpc":    "https://fetcher.indra.cloud/jsonrpc",
    "stream": "https://fetcher.indra.cloud/stream"
  },
  "capabilities": [
    {
      "method": "fetcher.get_metrics",
      "params_schema":
        "https://schemas.indra.cloud/fetcher.get_metrics.json"
    }
  ],
  "version": "1.2.0"
}
```

5 Evaluation Plan

No evaluation results are reported in this paper. The plan is to measure Indra along three axes, using tasks drawn from public tool-use and agent benchmarks, namely AgentBench [11], ToolBench [12], τ-bench [13], and UltraTool [14]:

1. **Tool use / API invocation**: percentage of correctly formed RPCs.
2. **Goal consistency**: cosine similarity between planned and realised subgoals.
3. **Structured-output adherence**: JSON schema compliance rate.

We also plan an open leaderboard tracking these scores monthly across frontier models; it is not built.

6 Security and Compliance

The design specifies a set of security primitives for data protection, auditability, and compliance, including RBAC for tenant isolation and OAuth 2.0 authentication, chosen to fit deployment under regimes such as GDPR. Table 1 lists them. None of these controls beyond schema validation exist in the reference implementation.

Table 1: Security primitives in the Indra design

Control	**Specified implementation**
Transport security	Mutual TLS, rotating certs
At-rest encryption	AES-256-GCM with HSM keys
Audit	Append-only trace, deterministic replay
Data sovereignty	Region-pinned storage; on-prem mode
Sandboxing	Firecracker micro-VMs, egress allow-lists
Authentication	OAuth 2.0 with JWTs; RBAC namespaces
Rate limiting	Leaky bucket per agent/user ID

7 Implementation Status

The reference implementation (`Five-Research/indra-mvp`) is a single-process Python package with three dependencies: the OpenAI client, Pydantic, and a JSON logger. It exists to show that a plan can be validated before it spends, that independent tasks can run in the right order, and that a failed task can be retried without losing the graph. It does not attempt the distributed system the earlier sections describe. Table 2 maps design to code.

Table 2: What the reference implementation covers, against the design

Component	Design (Sections 2–6)	Reference implementation
Queen	Managed service over a frontier model	Implemented. `Queen` class calling the OpenAI API; emits BeeScript; retries malformed plans
BeeScript	JSON DAG with goal, budget, subtasks	Implemented. Parser, validator (ids, dependencies, cycles, budget), executor that yields ready tasks in order
Scribe / Router	Rust/Actix service, WebSocket tracking, circuit breakers	Partial. Python `Router` using a file-backed queue; sequential execution of ready tasks; retry with backoff
A2A Gateway	FastAPI JSON-RPC gateway with schema validation, rate limits, trace chain	Not implemented
Workers	OCI containers with RPC sidecar; SLM and deterministic types	Partial. In-process `BaseWorker` subclasses with a registry decorator; two reference workers (travel, finance)
Compiler	Map-Reduce with LLM ranker and multi-format renderers	Partial. Aggregates worker result files into one JSON output; no ranker, no renderers
Credits	Gas/storage split, retry fees, timeout burns	Implemented in-process. Ledger with charges, retry fees, timeout penalties, refunds, per-session budget enforcement
Memory	Faiss index plus LSM, usage-weighted eviction	Partial. Per-session key–value store with count-based eviction; no vector index
Deterministic replay	Keccak hash chain, optional DA anchoring	Partial. Tasks and results persisted as JSON files; no hash chain
IVAM formal model	Section 3	Design only
Security primitives	Table 1	Not implemented
Evaluation	Section 5	Not run
Tests	—	Three: an end-to-end workflow, a BeeScript workflow, memory and credits

8 Related Work

Frameworks such as LangChain [3], AutoGen [4], AIOS [5], and Zapier Agents [15] offer agent composition but leave reliability, billing, and compliance to the application developer. AIOS is the closest in spirit, treating the LLM as an operating-system kernel; Indra differs in metering every call and in planning through an explicit DAG. Surveys on LLM multi-agent collaboration [6, 7] and adaptive team building [8] motivate the OS-level approach, while biology-inspired task allocation [9, 10] shaped the colony metaphor.

Table 3 compares Indra's design goals with the frameworks as we understand them at the time

of writing. The Indra column describes the design, not the reference implementation; the other columns rest on public documentation and third-party comparisons [16, 17] and should be read as a snapshot rather than a benchmark.

Table 3: Design goals of Indra against existing frameworks, at the time of writing

Feature	**Indra (design)**	**LangChain**	**AutoGen**	**CrewAI**	**LangGraph**
Deterministic replay for audits	✓	✗	✗	✗	Partial
Credit-based metering and budgets	✓	✗	✗	✗	✗
Parallel scheduling with latency bounds	✓	✗	Partial	✗	✓
Security primitives (mutual TLS, sandboxing)	✓	Partial	Partial	✗	Partial [17]
Persistent memory and goal context in the wire protocol	✓	✗	✗	Partial	Partial
Enterprise compliance and data sovereignty	✓	Partial	Partial	✗	Partial [16]

9 Future Work

The directions below are the plan; none is implemented yet.

1. **Edge-only mode**. All inference, memory, and routing on trusted hardware at the user's end, for environments where privacy comes first.
2. **Marketplace economics**. A two-sided marketplace where developers publish versioned Worker artifacts and consumers pay in credits, with surge pricing and reputation scores to align incentives and reward efficient agents.
3. **Ecosystem plug-ins**. Gateways that connect the swarm to closed SaaS or on-prem platforms (e.g. Salesforce, Bloomberg, Epic) through credential-sandboxed bridges.
4. **Anchored audit trail**. A hash of every transaction sent to an L2 data-availability layer for tamper-evident logs and verifiable settlement.
5. **Collective scheduling**. Allocating GPU time across tasks for maximum aggregate value under limited energy budgets.

10 Conclusion

Indra is a design for an orchestration layer that is open, auditable, and economically legible: every agent call is a metered, replayable transaction, and a plan cannot spend before it is validated. The reference implementation demonstrates the orchestration loop and the credit ledger; the distributed system around them remains a design. The lesson we take from building it is that the credit budget is both the product feature and the instrument that reveals whether the work is worth its cost. Pairing swarm-inspired coordination with OS-grade accounting is, we think, the right shape for systems in which more than one agent acts on a person's behalf.

References

[1] Asana. *Anatomy of Work Global Index 2023*. Asana, Inc., March 2023. https://asana.com/resources/anatomy-of-work

[2] Google. "Announcing the Agent2Agent Protocol (A2A)." Google for Developers Blog, 9 April 2025. Specification: https://a2a-protocol.org/latest/specification/

[3] H. Chase. *LangChain*. Software, 2022. https://github.com/langchain-ai/langchain

[4] Q. Wu, G. Bansal, J. Zhang, *et al.* "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation." arXiv:2308.08155, 2023.

[5] K. Mei, X. Zhu, W. Xu, *et al.* "AIOS: LLM Agent Operating System." arXiv:2403.16971, 2024.

[6] T. Guo, X. Chen, Y. Wang, *et al.* "Large Language Model based Multi-Agents: A Survey of Progress and Challenges." arXiv:2402.01680, 2024.

[7] E. La Malfa, G. La Malfa, S. Marro, *et al.* "Large Language Models Miss the Multi-Agent Mark." arXiv:2505.21298, 2025.

[8] L. Song, J. Liu, J. Zhang, *et al.* "Adaptive In-conversation Team Building for Language Model Agents." arXiv:2405.19425, 2024.

[9] A. Rajasekhar, N. Lynn, S. Das, and P. N. Suganthan. "Computing with the collective intelligence of honey bees—a survey." *Swarm and Evolutionary Computation*, 32:25–48, 2017.

[10] M. Kegeleirs and M. Birattari. "Towards applied swarm robotics: current limitations and enablers." *Frontiers in Robotics and AI*, 12:1607978, 2025. https://doi.org/10.3389/frobt.2025.1607978

[11] X. Liu, H. Yu, H. Zhang, *et al.* "AgentBench: Evaluating LLMs as Agents." arXiv:2308.03688, 2023.

[12] Y. Qin, S. Liang, Y. Ye, *et al.* "ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs." arXiv:2307.16789, 2023.

[13] S. Yao, N. Shinn, P. Razavi, and K. Narasimhan. "τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains." arXiv:2406.12045, 2024.

[14] S. Huang, W. Zhong, J. Lu, *et al.* "Planning, Creation, Usage: Benchmarking LLMs for Comprehensive Tool Utilization in Real-World Complex Scenarios." arXiv:2401.17167, 2024.

[15] Zapier. "Zapier Agents." 2024. https://zapier.com/blog/zapier-agents-guide/

[16] SuperAGI. "SuperAGI: Autonomous AI Agents Framework." 2025. https://superagi.com/

[17] AIMultiple Research. "Top AI Agent Frameworks Comparison." 2025. https://research.aimultiple.com/ai-agents/