Harness-First Agentic AI

Control Plane, Execution, and Governed Autonomy

AAIAAS (Agentic Artificial Intelligence as a Service) Reference Architecture Whitepaper

Version 1.1

July 2026

Table of Contents

1. Executive Summary

The prevailing narrative in enterprise AI focuses on models: which foundation model is best, which provider offers the best price-performance ratio, which prompt engineering technique unlocks the highest accuracy. This model-centric view produces systems that are fragile, opaque, and difficult to govern. When the model changes — and it will change — the entire system must be rebuilt.

The alternative is to flip the architecture. Place the model abstraction layer — the **Harness** — at the center of the design. Models become interchangeable commodities. Planning, policy, verification, and governance remain centralized and stable. Execution remains distributed, provider-agnostic, and resilient.

This whitepaper presents the reference architecture behind **AAIAAS (Agentic Artificial Intelligence as a Service)** — published at AAIAAS.ai — a platform built on this harness-first principle. It is designed for organizations that need autonomous AI that can be trusted: systems that execute reliably, remain compliant under pressure, and improve themselves over time.

The architecture delivers five capabilities that are rare in production:

1. **Provider-agnostic execution** — swap models without rewriting task logic
2. **Governed autonomy** — human-in-the-loop gates scaled across thousands of concurrent tasks
3. **Execution invariants** — circuit breakers, cost limits, and step ceilings that prevent runaway behavior
4. **Proof-of-execution** — every task produces tamper-evident audit artifacts
5. **Cognitive compounding** — skills that improve run-over-run through a closed self-improvement loop

The platform is deployed across three execution planes (cloud-shared, device-local, air-gapped), supports hybrid configurations where local inference assists without replacing central orchestration, and is designed to operate in sovereign or private environments where data residency requirements are strict.

This document describes the architecture, the governance model, and the operational invariants that make it work. It is intended for technical leaders, security architects, and operators evaluating autonomous AI platforms for production deployment.

Product-branded internal nomenclature and product-specific actor taxonomies are intentionally out of scope for this public reference architecture; they will be published when the control-plane product surface is mature enough for that level of detail.

2. The Problem: Model-Centric Failure Modes

2.1 Three Recurring Failure Patterns

Organizations that build autonomous AI systems around model capabilities — rather than around governance and control — encounter three predictable failure patterns:

Pattern A: Waiting. Recurring operational tasks that require a human to remember to execute them. Every Monday morning. Every Friday before audit. Every morning before standup. When the human forgets, the downstream process degrades silently. The task "works" in development but is fragile in production.

Pattern B: Silent Degradation. Automated processes that appear to function but whose outputs gradually drift from acceptable quality. No alert fires. No metric crosses a threshold. The degradation accumulates until

something downstream breaks, and the root cause is hours or days of stale outputs.

Pattern C: Talent Waste. Engineers and operators spending time maintaining brittle scripts, monitoring broken automations, and manually extracting compliance evidence. This is work that could be expressed in a sentence and executed autonomously, but is treated as infrastructure glue.

These are not niche problems. They are the reason recurring revenue exists in enterprise software. The organizations that solve them are not building better models — they are building better control.

2.2 Why the Model-Centric Approach Fails

The dominant paradigm treats the model as the product. Engineering effort flows into prompt engineering, model selection, context optimization, and output refinement. Governance is bolted on after the fact as a secondary concern.

This produces fragile systems because:

- **Model swap requires rewriting.** When a provider is deprecated, pricing changes, or a better model emerges, the task logic must be re-engineered because execution and model coupling are inseparable.
- **Governance is inconsistent.** Ad hoc safety checks and manual approvals do not scale. Teams that can't handle thousands of concurrent tasks with consistent risk-tiering cannot operate at production volume.
- **Audit trails are incomplete.** When every execution path is model-dependent and ad hoc, producing tamper-evident proof-of-execution for compliance is an engineering project, not a native capability.

The harness-first architecture inverts this priority. The model is a commodity input. The harness — the layer that provides provider isolation, verification, bounded execution, and governance — is the product.

---

3. Reference Architecture

3.1 Six Layers, Three Planes

The AAIAAS architecture is organized into six distinct layers, each with a bounded responsibility. This layering is not optional. Violating layer boundaries requires an architectural decision record and explicit review.

Layer	Responsibility
Experience Layer	User interfaces, dashboards, input capture
Orchestration Layer	Planning, routing, policy enforcement, task decomposition
Harness Layer	Model/tool abstraction, provider isolation, verification, governance
Capability Layer	Modular skills and connectors — provider-agnostic interfaces
Execution Layer	Workers that execute tasks on their assigned execution plane
Memory Layer	Tenant-segmented knowledge storage and retrieval

These layers operate across three execution planes:

Plane	Role
Control Plane	Strategic orchestration, planning, policy, and governance
Cloud Execution Plane	Shared multi-tenant worker infrastructure
Device Execution Plane	Tenant-owned local or on-premises workers

No plane may silently assume the responsibilities of another. Control plans and evaluates but does not execute. Execution workers execute but do not plan. This separation is the architectural invariant that makes governed autonomy possible.

3.2 The Harness Layer: Five Invariants

The Harness Layer is the differentiator. It provides:

Invariant 1 — Provider Isolation. Provider-specific execution logic is confined to dedicated adapters. Task logic (skills) never embeds model-provider concerns. Swapping a model provider requires changing the adapter, not rewriting the skill.

Invariant 2 — Provider-Agnostic Capabilities. Skill semantics are defined independently of any model. A "browse the web" skill produces the same output contract whether powered by a GPT model, a Claude model, or a local open-source model.

Invariant 3 — Graceful Degradation. Optional enrichment steps (retrieval-augmented generation, planner critic passes, billing telemetry) degrade gracefully unless explicitly marked as hard-required by policy. Failures produce structured diagnostics, not cascading failures.

Invariant 4 — Verification Before Completion. Task completion requires satisfaction of verification policies. A task does not complete until its output is verified against the acceptance criteria defined in the task spec.

Invariant 5 — Bounded Execution. Task graph expansion obeys explicit depth and fan-out limits. The system will not generate infinitely recursive task chains. Every execution path has a ceiling.

Six additional Harness invariants enforce safety hierarchy (the safety tier of the SQDEC ordering is inviolable, regardless of economic or delivery pressure), tenant-segmented memory, artifact-separation discipline (durable artifacts remain externalized and addressable, not collapsed into prompt context), and execution capability verification (the Harness verifies that the target execution plane can satisfy required skill contracts before admitting a task graph into the pipeline).

3.3 Execution Plane Invariants

The execution layer is governed by its own invariant set:

- **Plane Separation:** Control, cloud, and device responsibilities remain clearly delineated. No plane assumes the responsibilities of another.
- **Worker Traceability:** Every worker maintains a verifiable identity including worker ID, execution plane assignment, and authorized tenant set.
- **Proof-of-Execution:** Every completed task emits proof artifacts including execution metadata, artifact hashes, and verification signals.
- **Fail-Closed Execution:** Workers fail closed on unsupported skill types, invalid payload schemas, or missing authorization. Silent fallback execution is forbidden.

- **Tenant Authorization:** Worker authorization is evaluated against the worker's authorized tenant set. Device workers serve a single tenant. Cloud workers serve an authorized set. Cross-tenant execution is always auditable.

3.4 Tenant Isolation

Every tenant undergoes a bootstrap lifecycle before activation: a defined provisioning sequence that establishes tenant record, settings, feature flags, memory namespace, public skill access, execution authorization, and observability scope. A tenant must not reach active status until all bootstrap resources exist. Authentication success alone does not imply readiness.

All tenant data and execution remain isolated. Cross-tenant data leakage is structurally forbidden by the architecture.

---

4. Control Plane and Worker Governance

4.1 Roles and Responsibilities

The platform separates orchestration from execution:

Control-plane orchestrator. Receives intent, expands goals into task specifications, generates execution plans, and exercises human-in-the-loop approval authority. Orchestration logic lives in the control plane — never in the execution layer.

Worker agents. Execute tasks dispatched by the control plane on an assigned execution plane. Workers are capable and precise, but act only under control-plane direction. A worker that cannot report status or accept recall is treated as a system failure.

Staging environment. Local development and staging is where workers are trained, skills are developed, and configurations are tested before production deployment.

Operator dashboard. The elevated observation and decision surface where operators monitor task state, review HITL approval queues, and direct execution.

4.2 Governance Controls

Three mechanisms keep autonomous execution safe:

HITL gate. Tasks at the High and Critical risk tiers are held behind a human approval gate. No task bypasses this silently. Approval is never removed by default — it requires explicit human action (approve, reject, or waive-by-action at the individual task level).

Budget constraints. Every worker has an associated cost envelope. Execution paths are bounded by per-task and per-session spending limits. When the envelope is exhausted, execution halts and reports — it does not continue silently.

Control-plane tether. Workers remain connected to the control plane through heartbeat signals and recall paths. Even workers running locally with local inference assistance are tethered to the central orchestrator. They can operate semi-independently but can always be recalled.

4.3 Telemetry and Observability

Every worker emits telemetry at every lifecycle stage — planning, dispatch, execution, completion, and verification. Silent operation is not an option. Every task produces structured logs suitable for aggregation and analysis, with traceability across tenant, worker, execution plane, and skill.

5. HITL Governance: Risk-Tiered Human-in-the-Loop

5.1 Risk Tiers

The platform classifies every operation into one of four risk tiers, and each tier has a defined approval requirement:

Tier	Definition	Default Action
Low	Routine, idempotent, or read-only operations with no security or cost impact	Auto-approve
Medium	State-changing operations with limited scope or reversible impact	Notify only
High	Critical state changes, significant cost implications, or security-sensitive actions	Require approval
Critical	Irreversible destructive actions or root-level security changes	Require typed confirmation

5.2 Operation Permanence Classification

Operations are further classified by permanence:

- **Type 1 (Reversible):** Operations where the system state can be restored via automated rollback or point-in-time recovery. These require a documented rollback plan in the task metadata but may proceed under medium-tier rules.
- **Type 2 (Irreversible):** Operations that result in data loss, irreversible state changes, or non-refundable costs. These always require HITL approval regardless of risk score and must include explicit risk acceptance.

5.3 Waiver Policy

Waivers are granted per-action only. No session-wide bypass is permitted. No silent bypass exists — every waived HITL checkpoint generates an auditable event. System-level invariants — specifically the safety constraints of the SQDEC ordering — cannot be waived under any circumstances.

High and Critical risk operations cannot be approved via voice interfaces. Critical risk operations require typed confirmation to prevent accidental activation.

5.4 High-Risk Operation Catalog

The following operation categories are hard-coded as High or Critical risk:

- Deploy to Production: **High**
- Delete Data (Database or Storage): **Critical** (Type 2)
- Billing Changes (Plan upgrade, downgrade, or cancellation): **High**
- Secret Rotation or Credential Invalidation: **High**
- Skill Promotion (Development to Production): **High**

This catalog is not exhaustive — any new operation category is evaluated against the same risk framework, but these are the established baseline for the platform's operating posture.

6. OGACS: Execution Invariants and Operational Governance

6.1 What Is OGACS?

OGACS (Operational Governance for Autonomous Cognition Systems) is the minimal governance layer that enforces execution invariants in real time. It runs alongside the runtime and intercepts operations as they occur, evaluating each against a fixed set of invariants before allowing execution to proceed.

OGACS is not a policy engine that loads rules dynamically. It is an invariant enforcement layer with a fixed, auditable set of constraints. Its role is to prevent drift — the divergence between intended and actual execution behavior.

6.2 Enforcement Model

OGACS evaluates operations through two primary dimensions:

Operational Mode. The system operates in one of two modes:

- **OPEN mode** (standard): All capabilities are available, standard governance applies. This is the default operational condition.
- **LOCKDOWN mode** (containment): Execution is suppressed; only safety-critical operations are permitted, and all other operations require explicit operator authorization. Skill loading collapses to the safety layer only.

Drift Detection. The system continuously compares current operational state against the authoritative baseline (the committed configuration state). When drift is detected, OGACS triggers convergence workflows — controlled processes that align the current state back to the authoritative baseline through merge, re-commit, or governance intervention.

6.3 Invariant Set

OGACS invariants cover the following domains:

- **Approval Gates:** Operations at defined risk thresholds require human authorization before execution. No bypass.
- **Step Limits:** Each execution path has a maximum step count. The system will not execute beyond the limit.
- **Cost Limits:** Per-task and per-session cost ceilings prevent runaway spending.
- **Circuit Breakers:** When error rates, timeout rates, or quality degradation exceed thresholds, the system halts execution and escalates to human review.
- **Tenant Segregation:** Cross-tenant operations are forbidden unless explicitly authorized. Isolation is structural, not policy-dependent.
- **Artifact Separation:** Durable outputs remain externalized and addressable. Prompt context and execution memory never replace artifact records.

6.4 Drift and Convergence

When the system detects that current state diverges from the authoritative baseline, it initiates a convergence workflow:

1. **Detection:** State comparison against committed baseline identifies the divergence.

2. **Classification:** The drift is classified by severity (minor configuration change, skill version mismatch, execution policy deviation).
3. **Resolution:** The system either automatically converges (for minor changes) or escalates to human review (for significant deviations).
4. **Verification:** Post-convergence validation confirms the system is back to authoritative state.

This loop is continuous and automated. Drift detection is not a periodic audit — it is a real-time structural property of the architecture.

---

7. Cognitive Compounding: Self-Improving Skill Loops

7.1 The Evolutor Pattern

Most AI platforms deliver static capability: you define a skill or workflow, it executes as designed, and improvement requires manual re-engineering. The AAIAAS platform implements a cognitive compounding pattern — a self-improvement loop where skills become more effective through accumulated execution experience.

This is not training in the traditional ML sense. The platform does not fine-tune models or adjust weights. Instead, it implements a skill evolutor that observes execution outcomes, evaluates quality signals (completion rate, verification pass rate, operator feedback, cost efficiency), and proposes skill improvements for operator review.

7.2 How It Works

The evolutor pattern operates on three timescales:

Run-over-run improvement (immediate). Each execution produces structured quality signals — did the task complete? Did it pass verification? How much did it cost? How many steps were required? These signals accumulate into a skill health score that is visible at a glance. A health score of 82/100 with three specific recommendations tells the operator exactly what to improve.

Skill iteration (weekly). Skills that consistently score below a threshold trigger an improvement cycle. The system proposes modifications — adjusted parameters, alternative execution paths, different enrichment steps — and the operator reviews and approves changes. Approved changes are versioned and tested before promotion to production.

Cross-skill learning (monthly). Patterns that emerge across skill executions — such as a particular verification step consistently catching the same class of errors — are abstracted into general improvements that propagate to related skills. This is where compounding accelerates: each skill's improvement raises the floor for all others.

7.3 Why This Matters

The cognitive compounding pattern produces a system whose capabilities grow over time without proportional engineering investment. The first run of a skill delivers baseline value. The twentieth run, informed by accumulated execution data, delivers measurably better results. The hundredth run, benefiting from cross-skill patterns, delivers results that no single engineer could have engineered manually.

This is the core product thesis: agents that improve themselves are not a feature — they are the only architecture that scales autonomously without linearly scaling engineering overhead.

8. Execution Planes: Cloud, Device, and Air-Gap

8.1 Cloud Execution: Multi-Tenant Worker Infrastructure

The cloud execution plane provides shared, multi-tenant worker infrastructure hosted on managed cloud services. Workers in this plane:

- Serve multiple tenants simultaneously
- Are managed and upgraded centrally
- Benefit from shared model provider relationships and caching
- Require an authorized tenant set for cross-tenant access
- Always maintain the control-plane tether — heartbeat, registration, and recall paths

This is the default deployment posture and the operational baseline. Most tasks execute in the cloud plane.

8.2 Device Execution: Tenant-Local Workers

The device execution plane provides tenant-owned local or on-premises workers. Workers in this plane:

- Serve a single tenant exclusively
- Run on infrastructure controlled by the tenant
- May operate with local inference models (Ollama, local LLMs) for preprocessing, retrieval, summarization, and redaction
- Maintain the control-plane tether — they can operate semi-independently with local-assist inference but remain connected to the central orchestrator
- Support hybrid configurations where local inference assists strategic planning without replacing it

Local inference in device mode is explicitly constrained: it operates as a **Harness Layer co-processor**, not as a second orchestrator. It may assist with local retrieval, summarization, fact extraction, redaction, and privacy-preserving preprocessing. Strategic planning, policy decisions, and authoritative task orchestration remain control-plane responsibilities.

This separation is structural. The control plane is the brain. Device-local models are the sensory apparatus.

8.3 Air-Gap Deployment

For tenants with strict data residency or sovereignty requirements, the architecture supports air-gap execution:

- No outbound network connectivity from the execution plane
- Workers execute entirely within the tenant's network boundary
- The control plane may be deployed within the same network or accessed through a one-way data diode
- Model inference runs on local hardware
- All artifacts, proofs, and audit logs are generated and stored on-premises

This is the most restrictive deployment mode and requires careful operational planning, but it is natively supported by the architecture, not bolted on as an afterthought.

8.4 Hybrid Configurations

Most production deployments will use hybrid configurations:

- Strategic planning runs in the cloud control plane
- Data-sensitive preprocessing runs on local workers with local models
- Heavy computation tasks run in the cloud plane
- Compliance-critical verification runs on-premises
- The control-plane tether connects everything, ensuring that even locally operating workers can be directed, recalled, and audited by the central orchestrator

Local-assist workers may operate semi-independently in the field while remaining within tether distance and subject to recall at any time.

---

9. Deployment Topologies and Production Posture

9.1 Canonical Deployment Reference

The platform supports the following deployment topologies:

Production Cloud (default). The control plane is deployed on a managed cloud service (Railway) with a backing database (Supabase). Workers run in the cloud execution plane. Local device workers are optional. This is the standard configuration for most tenants.

Sovereign/On-Premises. The control plane is deployed within the tenant's infrastructure. Model providers may be external (via secure API) or fully local. Workers run on tenant hardware. This configuration satisfies strict data residency and sovereignty requirements.

Hybrid. The control plane runs in the cloud. Device workers run on-premises or at the edge. The two communicate over encrypted channels with the control-plane tether. This is the default posture for enterprise deployments with distributed operations.

This whitepaper is published by AAIAAS.ai (Agentic Artificial Intelligence as a Service). Architectural invariants, governance patterns, and operational procedures described herein reflect the public reference architecture. Product-specific naming systems and actor taxonomies are reserved for later publication when the control plane is ready.