MCP OS: A Policy-First Capability Environment for Model-Neutral AI Agents

Version 1.1

Charles E Morgan IV
Independent Researcher

July 29, 2026
Repository evidence: `33154ee77b2d3447613a20a9c4cde1c9171a9ed9`

Abstract

The practical value of contemporary AI agents depends less on a model's ability to generate text than on its ability to interact safely with real systems. The Model Context Protocol (MCP) standardizes an important portion of that interaction: connections from AI applications to tools, data sources, and workflows. It does not, by itself, supply a full operational environment for policy, approvals, durable task state, auditability, model substitution, or enterprise governance. This paper describes MCP OS, a TypeScript capability-runtime foundation designed to occupy that intervening layer. The implementation uses model- and transport-neutral contracts; deny-by-default policy evaluation; scoped, expiring approvals; a closed task-state graph; local audit events; versioned MCP-tool normalization; guarded Project, Filesystem, and Git capabilities; and verified local-Ollama demonstrations. It also documents a signed, loopback-only macOS companion beta whose production MCP surface is one policy-governed tool. The paper distinguishes that bounded beta—including policy-gated Terminal proposals, paired authority, receipts, Stop, and operator support diagnostics—from unqualified all-application control, remote operation, and enterprise identity. The resulting contribution is an architectural argument rather than a claim of production completeness: agentic systems need a stable capability environment between changing reasoning engines and high-consequence operational tools.

Keywords: AI agents, Model Context Protocol, governance, policy enforcement, capability runtime, enterprise AI, tool interoperability

1 Introduction

The center of gravity in applied artificial intelligence is moving from short, self-contained conversations toward delegated tasks that invoke tools, inspect environments, and run over longer time horizons. OpenAI reports that sampled Codex users increasingly request tasks that would take more than an hour of human work, while adoption has expanded beyond engineering into business functions [2]. This shift creates a systems problem. A model can reason about a task, but useful work often requires access to source repositories, files, services, identity systems, browsers, devices, or databases. Each access path introduces authorization, safety, observability, and recovery requirements.

MCP is a major step toward interoperable tool access. The protocol defines an open standard that lets AI applications connect to external data, tools, and workflows [1]. Yet a standard connection surface is not an operating environment. A production deployment must also decide which

capability may run, under which identity, with what approval, within which task, and with what audit record. In parallel, enterprise identity products are beginning to treat agents as governed principals with lifecycle and access-management needs [3]. These developments motivate a layer that is neither a model provider nor merely a server catalog.

MCP OS is an early implementation of that layer. Its premise is that models and MCP servers should be replaceable at the edge while a capability environment preserves policy, approval, task-state, audit, and normalization boundaries. The system is deliberately incomplete. Its value as a case study lies in showing how the governance layer can be designed before broad operational access is granted.

2 Method and Evidence Boundary

This version is a systems analysis of companion commit `33154ee` (July 29, 2026); its full 40-character identifier is recorded in the repository citation. Market context is drawn from primary sources. Implementation claims are based on source code, tests, architecture documents, threat model, user and test guides, release-acceptance material, and the local-Ollama beta guide [5, 6, 8, 9, 10, 12, 13, 14]. The analysis classifies functionality as *verified* when it exists in the present codebase and is covered by repository tests or direct source inspection; *verified beta* when it has bounded installed or recorded local evidence but is not validated as a production service; and *approved design* when an agreed architecture exists but the corresponding package or qualification is not executable.

This distinction is material. The present implementation includes a local-Ollama path, a bounded terminal runner, and a signed packaged loopback companion beta with paired clients, task-scoped authority, receipts, Stop, and a one-tool production catalog. It does not establish durable enterprise storage, remote operation, enterprise identity, a complete suite of capability servers, or empirical qualification of general control across all installed applications. The figures and market analysis therefore describe an architectural position and an implementation trajectory, not a claim that all surrounding layers are deployed.

Evidence layer	**Observed scope at v1.1**	**Interpretation**
Source verification	Lint, type-check, 1,338 unit tests (6 skipped), 14 integration tests, and a 14-package build passed at the cited commit.	Verified source
Installed beta	The signed loopback companion has an acceptance procedure covering listener, pairing, authority, rejection, Stop, audit, and follow-up observation.	Verified beta boundary
General-control qualification	The five-executor ledger requires ten clean, proof-backed samples per scenario and is marked *not run*.	Not a released claim

Table 1: v1.1 evidence boundary. The source suite does not substitute for installed or general-control qualification.

3 Background: From Agent Tools to Capability Environments

3.1 MCP and the interoperability problem

MCP addresses fragmentation in the way AI hosts reach tools. The official introduction compares the protocol to a universal connection point: an AI application can connect to a local file system, a database, a search engine, or a specialized workflow through a shared approach [1]. This is valuable because agents otherwise need bespoke integration code for every provider-tool pairing. MCP's abstraction, however, is intentionally narrower than organization-wide policy or identity management.

That boundary matters in an enterprise setting. A tool may be technically discoverable yet still be inappropriate for a particular agent, task, user, time window, or input. The governance requirement grows with capability consequence. Reading an allowlisted project summary is not equivalent to staging a Git change, deleting data, or transmitting customer information. Consequently, a tool protocol benefits from a companion control plane that can express risk, decisions, approvals, and audit semantics independently of a particular model or server implementation.

3.2 The emerging agent-governance market

The market can be understood as several complementary layers rather than a single winner-take-all category. Model providers compete on reasoning, latency, cost, and modality. MCP and related interfaces supply tool interoperability. Identity and security providers govern principals, access rights, and lifecycle. Coding agents and IDEs package agentic interaction into developer workflows. A capability environment sits between those layers, translating an agent's requested action into a policy-governed, auditable execution path. Microsoft Entra's agent identity documentation illustrates the surrounding market direction: agent identities can be created, sponsored, governed, and granted access in an enterprise lifecycle [4].

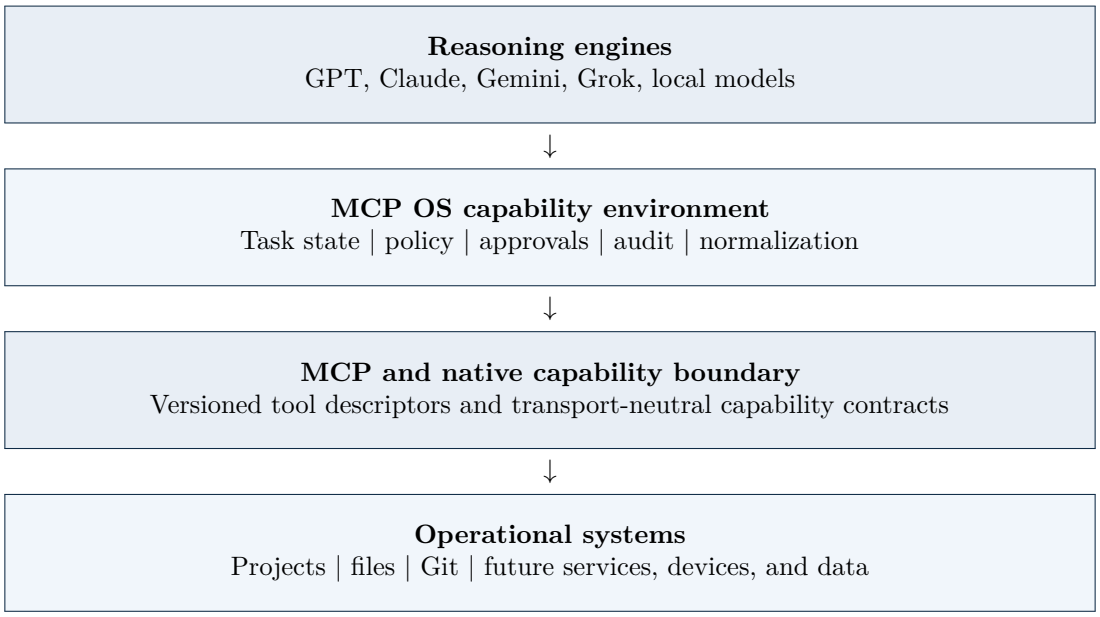

Figure 1: The proposed capability-environment position. The middle layer remains stable as models and operational connectors change.

4 System Architecture

4.1 Foundation components

MCP OS is organized as a pnpm and TypeScript monorepo. Its Phase 1 foundation establishes shared contracts, configuration, persistence seams, audit services, and a deterministic mock model provider. Phase 2 adds policy, approvals, task state, and the agent runtime. Phase 3 completes a narrow, guarded set of local capability services and adds a local-Ollama beta runner. The architecture document states the governing principle directly: model and capability contracts are provider- and transport-neutral [7].

The system separates responsibilities into focused packages. Shared types represent normalized contracts. Configuration validates application and model settings. Persistence is represented by interfaces rather than a production database dependency. Audit stores append-only events through a narrow service boundary. The model gateway establishes a normalized provider contract, while the policy engine and agent runtime provide governance and lifecycle behavior. This structure avoids allowing raw provider payloads or raw MCP protocol payloads to dictate the runtime's internal model.

4.2 Task state and approval

The runtime defines a closed lifecycle covering task creation, planning, approval, tool execution, synthesis, completion, failure, and cancellation. Its state machine rejects transitions that are not present in the graph. The minimal in-memory runtime can submit a task for approval or straight to completion, supports resumption only from the approval state, rejects resumption for other states, and refuses cancellation of a terminal task. Audit events capture task creation and cancellation in the present foundation.

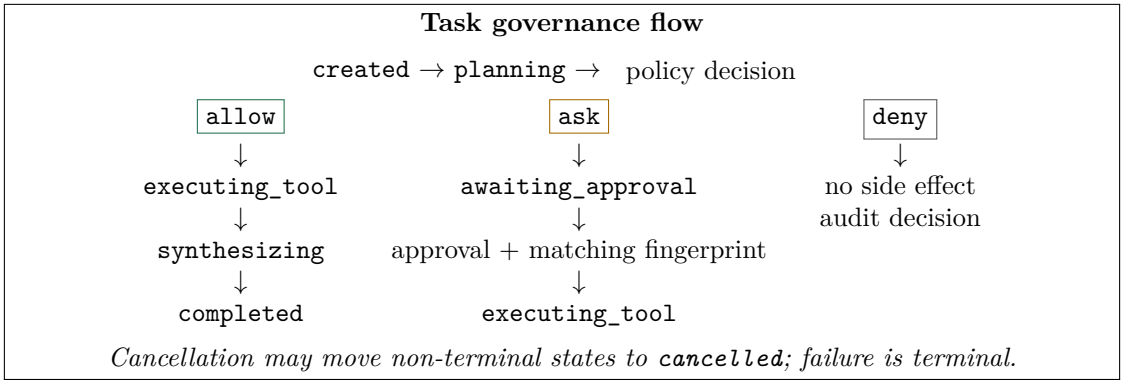

Figure 2: The policy and task-state model. Approval is an explicit state, not an informal user-interface convention.

The approval service is designed to bind a consent decision to a narrow context. An approval has an identifier, task identifier, input fingerprint, expiration, and approval status. Consumption fails if the approval is absent, unapproved, expired, or associated with a mismatched fingerprint. This is a small but important design choice: approval of one proposed action should not silently authorize a different action.

4.3 Policy engine

The policy engine evaluates rules with three effects: `allow`, `ask`, and `deny`. Rules have priority and may target a capability identifier. The engine filters matching rules, orders them by descending priority, and selects an explicit denial before considering the highest-priority remaining match. If no rule matches, it returns `deny`. This makes the default posture conservative and allows a specific prohibition to override a broader permission. The present implementation is intentionally small: it establishes the semantics that later policy inputs, identity systems, and capability services can consume.

4.4 MCP normalization and capability services

Phase 3 adds a separate MCP client package. Its tool adapter accepts a descriptor containing a server identifier, tool name, version, description, input schema, risk level, and side-effect declaration. It validates server and tool identity characters, then produces a normalized capability identifier in a versioned form such as `mcp.server.tool-vN`. Execution is represented as an internal function accepting an unknown input and delegating to an invoker. The key architectural result is that the runtime can reason about a common capability shape without depending on raw MCP payloads.

Tool-normalization boundary

MCP descriptor	Boundary	Normalized capability
serverId, name, version, description	identity validation and versioned naming	id, description, `execute(input)`

Example: server id git, tool name status, version 1 becomes a versioned internal capability id.

Figure 3: MCP tools are adapted at the boundary rather than passed as protocol-specific objects into the capability runtime.

The connection manager separately registers a loader for each server and records a health state. A successful tool listing marks a server healthy; a loader failure marks it unhealthy and returns an empty tool list. It applies a bounded timeout, supports reconnect, rejects duplicate normalized capability identifiers, and emits audit callbacks. This failure isolation avoids presenting a temporarily failed server as a successful discovery result. It is not a complete remote MCP transport or a universal discovery system.

Three guarded capability services extend the architecture. The Project service enforces an allowlist and reports bounded tree, package, dependency, and configuration summaries. The Filesystem service canonicalizes paths, blocks traversal and symlink escapes, bounds reads and searches, redacts credential-like paths, and returns metadata. The Git service exposes bounded read operations and requires a scoped approval for branch creation, staging, commits, and pushes; force pushes, resets, deletions, cleanup, and arbitrary shell execution are unavailable. Configuration examples and cross-server security tests support these boundaries [9]. These services are deliberately narrow interfaces, not complete production interfaces.

4.5 Local-Ollama beta demonstration

The repository also includes a terminal demonstration that connects a locally installed Ollama model to a disposable fixture repository. The runner asks the model to inspect a generated project through a small allowlisted tool set, then requires an explicit terminal approval before

invoking the Git branch-creation capability. In a recorded local run using `qwen3.5:9b`, the model completed `project_summary`, `read_package`, and `git_status`; after approval, it created `demo/approved-change`. The audit timeline recorded readiness, fixture creation, model responses, each tool completion, the approval request and grant, and completion [10]. This is evidence of an end-to-end beta path, not a benchmark or a claim that arbitrary models, repositories, or tools are production-ready.

4.6 Persistent local companion beta

The current beta packages MCP OS as a login-started, loopback-only macOS companion rather than embedding governance in a particular frontend. A native helper supplies the narrow macOS boundary for screen observation, application activation, mouse and keyboard input, permission diagnosis, and cancellation. The companion owns the higher-order control plane: client pairing, policy evaluation, immutable per-run authority, local audit and receipts, evidence retention, bounded operator support diagnostics, and an operator-facing Stop control [11, 13].

The production MCP catalog exposes one model-facing entry point, `mcp_os_execute`. It creates or continues a task-scoped run, executes a typed computer, browser, Terminal, Project, Filesystem, or Git request, reports status, and stops the run. Raw executor, map-administration, and diagnostic tools are development-only. The unified gateway routes an allowed typed request to bounded native input, mapped semantic GUI, browser, terminal, Project, Filesystem, or Git execution; it requires request-bound proof before reporting completion [12]. This is a deliberately smaller production surface than the internal capability inventory.

The pairing model separates client identity from action authority. Every new client begins `read_only`; the operator selects `approve`, `delegate`, or `full` for a particular run, and a model cannot elevate that authority. Terminal requests are classified by MCP OS and await matching approval when policy requires it. Input actions require a fresh, run-bound observation that matches display geometry and foreground context. Stop cancels pending work and fences subsequent input. This is a completed packaged beta boundary, not evidence of general all-app task completion.

5 Security and Governance Analysis

The system's security model is most accurately understood as progressive exposure. The threat model establishes schema validation at public boundaries, avoids logging configuration secrets, and represents audit records through append-only persistence interfaces [8]. Phase 2 adds deny-by-default policy and approval behavior before side effects. Phase 3 protects the local-capability boundary with scoped allowlists, canonical filesystem path enforcement, bounded and redacted output, duplicate-capability rejection, connection health reporting, bounded Git reads, and approval-gated Git writes. Destructive Git operations and arbitrary shell execution remain unavailable.

This separation is valuable for two reasons. First, it prevents a policy-free tool catalog from becoming the de facto runtime before governance semantics are defined. Second, it makes residual risk visible rather than hiding it behind product language. The companion stores captures below an owner-only local data root, rejects existing or symlinked capture destinations, and gates protected effect-capable applications even under delegated authority [8]. MCP OS does not claim to eliminate prompt injection, identity misuse, or server compromise. It also does not claim isolation from another malicious process already running as the same macOS user; a hardened public release would need a caller-validated native boundary.

The connection between agent capability and identity must remain explicit. Microsoft frames agent identities as principals that should be managed and governed throughout an agent lifecycle [4].

MCP OS does not yet integrate an identity provider. It can nevertheless serve as an architectural seam: identity can produce a trusted execution context, policy can evaluate a capability request against it, approval can bind the resulting action, and audit can preserve the decision trail.

6 Market Position

MCP OS is not positioned as a replacement for reasoning engines, identity platforms, IDEs, or the MCP standard. Its market position is the coordination layer that can compose those categories without collapsing them into one vendor-specific surface. Figure 4 maps this role by primary value proposition. The map is qualitative: it does not score specific vendors or infer undisclosed product features.

Qualitative market map

Reasoning and end-user work	Control and governance
Model providers and agent products Model quality, interaction, task completion, and user experience.	**Identity and security platforms** Authentication, lifecycle, access governance, and enterprise controls.
Tool protocols and connector ecosystems Standardized discovery and invocation across tools, data, and workflows.	**MCP OS capability environment** Policy, approval, task-state, audit, and normalized capability execution between changing agents and operational systems.

The categories overlap in practice; the figure identifies the primary architectural center of gravity.

Figure 4: MCP OS occupies the capability-environment position between interoperability and enterprise governance.

This position has three potential advantages. First, it can support model substitution: a capability is represented internally rather than coupled to a particular model provider's tool representation. Second, it can support tool substitution: a normalized capability contract can outlive a particular MCP server or native implementation. Third, it can centralize governance semantics without assuming that one vendor must own every model, connector, device, and identity system. These are design advantages, not yet measured economic advantages.

The strategic challenge is equally clear. The layer is valuable only if it becomes operationally trusted. That requires reliable identity integration, durable storage, observable policy decisions, capability-level security hardening, and usable administration. The current codebase supplies some governance primitives but does not yet demonstrate the scale, integrations, or operational evidence needed to substantiate a production-enterprise claim.

7 Implementation Maturity and Limitations

Figure 5 summarizes the maturity boundary. The chart intentionally uses categorical status rather than fabricated coverage percentages. It makes the paper's main qualification visible: the foundation is real, but the broader platform remains incomplete.

Several limitations follow directly. The agent runtime and companion evidence stores are local rather than an enterprise durability service. The policy engine does not yet demonstrate attribute-rich enterprise policy evaluation. The companion is deliberately loopback-only and does not pro-

System area	Evidence-grounded scope	Status
Core contracts, policy, approvals, task state	Model-neutral contracts; deny-by-default rules; scoped approval; closed lifecycle	Verified
Audit and persistence	Foundation seams plus companion-local audit, receipt, and evidence retention	Verified beta
MCP client boundary	Versioned descriptors; timeout/reconnect; collision rejection; health and audit callbacks	Verified beta
Project, Filesystem, Git services	Allowlists; canonical and redacted filesystem access; bounded Git reads; approval-gated writes	Verified beta
Local model demonstrations	Ollama fixture runner and bounded macOS computer-control beta	Verified beta
Persistent local companion	Signed loopback beta, pairing, one production tool, local audit/receipts, Stop, and bounded operator support diagnostics	Verified beta
General-control qualification	Five executor scenarios require ten clean proof-backed samples each; the release ledger is not run	Unqualified
Production platform layers	Durable storage, broader server coverage, remote operation, and enterprise identity integration	Internal 1.0/1.1 design

Figure 5: Implementation maturity classification derived from the repository source, threat model, and Phase 3 plan.

vide remote operation, OAuth, multi-user tenancy, or enterprise identity. Its one-tool surface is a bounded beta: completion requires executor evidence bound to the request, but that contract alone is not an empirical claim of arbitrary-app reliability. The five-executor release ledger remains unrun, requiring ten clean samples for each of Calculator, TextEdit, a loopback Chrome fixture, unsent Mail, and a fixed Numbers adapter. Consequently, neither a source suite nor historical installed acceptance evidence may be represented as a general all-app or performance result. The Ollama runners remain proofs of concept over a generated fixture and bounded desktop workflow, not benchmarks of model reliability, repository safety, throughput, user experience, or enterprise security effectiveness.

These limitations should not be treated as incidental. They define the appropriate interpretation of MCP OS: a governed capability-runtime foundation. The platform thesis is plausible only if later phases preserve the present boundary discipline while adding robust authentication, authorization, storage, observability, and connector behavior.

8 Implications and Future Work

The architectural lesson is that agent infrastructure benefits from separating reasoning from authority. Models can change quickly. Tool protocols can improve. Local and cloud execution environments can coexist. The stable system boundary should be the policy-governed capability request: who or what requested it, what capability is implicated, what input is proposed, what decision applies, whether approval is required, and what event record results.

Near-term work follows from the remaining qualification gap. MCP OS should execute the

existing five-executor contract in a dedicated operator-approved lab: each scenario must produce its own fresh observations, MCP results, audit events, receipts, final proof, and cleanup proof before it contributes to an aggregate result. The fixture coordinator must remain internal, never become a customer-facing MCP tool, and prepare only reviewed foreground, reset, and loopback browser state. Capability services should continue to be exercised against representative repositories and operational back ends while retaining their narrow allowlists and approval gates. Remote transport, cloud evidence, and enterprise identity integration remain deferred internal 1.0/1.1 work rather than implicit consequences of the local companion.

Research evaluation should then shift from structural correctness to empirical evidence. Useful metrics include policy-decision latency, approval completion time, rate of denied unsafe operations, audit completeness, recovery from connector failure, model-switching friction, end-to-end task success, and operator comprehension of a proposed action. Comparative work should distinguish a capability environment from both generic agent frameworks and tool-protocol implementations.

9 Conclusion

MCP makes it easier for AI applications to reach tools and data. It does not by itself decide when that reach should be exercised. MCP OS is an early effort to make that decision layer explicit. At v1.1, its implementation establishes model-neutral contracts, deny-by-default policy, scoped approvals, task-state foundations, audit seams, guarded capability services, a local-Ollama beta path, and a signed loopback companion beta. The companion exposes one public MCP entry point that mediates typed requests through policy, task-scoped authority, execution receipts, request-bound proof, and Stop. Its market position is therefore not "another model" or "another MCP server," but a policy-first capability environment between agent reasoning and operational systems.

The work remains incomplete, and v1.1 preserves that qualification. The verified beta demonstrates a bounded local companion and a constrained, audited, approval-gated tool path; it does not establish production readiness or general all-app task completion. The unrun five-executor ledger is an explicit release gate, not a missing footnote. The broader opportunity is credible only if the implementation advances with durable, identity-aware, observable, securely bounded behavior and empirical qualification that reports both successes and failures.

References

[1] Model Context Protocol. (2026). *What is the Model Context Protocol (MCP)?* `https://modelcontextprotocol.io/docs/getting-started/intro`

[2] OpenAI. (2026, June 25). *How agents are transforming work*. `https://openai.com/index/how-agents-are-transforming-work/`

[3] Microsoft. (2026). *Microsoft Entra Agent ID documentation*. Microsoft Learn. `https://learn.microsoft.com/en-us/entra/agent-id/`

[4] Microsoft. (2026, June 16). *Governing agent identities*. Microsoft Learn. `https://learn.microsoft.com/en-us/entra/id-governance/agent-id-governance-overview`

[5] MCP OS. (2026). *MCP OS source repository* (Companion commit `33154ee77b2d3447613a20a9c4cde1c9171a9ed9`, July 29) [Computer software]. Private repository.

[6] MCP OS. (2026). *Telethryve Capability Bridge: Design Scope* [Unpublished design document]. `docs/DESIGN_SCOPE.md`.

[7] MCP OS. (2026). *Architecture* [Unpublished architecture document]. `docs/ARCHITECTURE.md`.

[8] MCP OS. (2026). *Threat Model* [Unpublished threat-model document]. `docs/THREAT_MODEL.md`.

[9] MCP OS. (2026). *Phase 3 remaining work and release checklist* [Unpublished implementation document]. `docs/PHASE_3_REMAINING.md`.

[10] MCP OS. (2026). *Ollama beta demonstration guide* [Unpublished demonstration document]. `docs/OLLAMA_DEMO.md`.

[11] MCP OS. (2026). *Persistent local companion design* [Unpublished design document]. `docs/superpowers/specs/2026-07-17-persistent-local-companion-design.md`.

[12] MCP OS. (2026). *MCP OS user guide* [Unpublished operator document]. `docs/USER_GUIDE.md`.

[13] MCP OS. (2026). *MCP OS Companion beta test guide* [Unpublished test document]. `docs/COMPANION_TEST_GUIDE.md`.

[14] MCP OS. (2026). *Five-executor beta evidence ledger* [Unpublished qualification ledger]. `docs/release/five-executor-beta-evidence.md`.