ActPass

CONFIDENTIAL

TECHNICAL WHITE PAPER

A Deterministic Authorization Gateway and Evidence Fabric for AI Agents

Deterministic allow / deny for AI agents — signed, replay-proof, audit-ready.

ActPass sits between your AI agents and their tools: every risky action gets a deterministic, signed allow / deny / require-approval decision — rendered outside the model — and a tamper-evident audit trail.

- Decision outside the model
- Signed Action Passports
- Tamper-evident evidence
- Fail-closed by design

CONFIDENTIAL — FOR DESIGN PARTNERS

Version 0.9 · Pre-GA · 2026-07-04

www.actpass.org
Managed authorization for AI agents

ActPass: A Deterministic Authorization Gateway and Evidence Fabric for AI Agents

Technical White Paper
Confidential — For Design Partners

Version 0.9 · Pre-GA · 2026-07-04

Abstract. Autonomous AI agents hold a human's standing authority — credentials and tools — and exercise it at machine speed. Prompt-level guardrails cannot be an authorization boundary: a probabilistic control that can be talked into a $50,000 refund once can be talked into it again, and a decision you cannot reproduce you cannot audit or defend. ActPass moves the decision out of the model. It is a deterministic, fail-closed authorization gateway and tamper-evident evidence fabric that sits between an agent and the tools it calls. For every risky action, a pure server-side function renders `allow` / `deny` / `require-approval` over a JSON policy DSL with first-match-wins rules, emitting one of 60 typed reason codes and defaulting to deny. Five ideas are load-bearing: the decision lives outside the model; fail-closed by mandate; no client-trusted fields; the API key is the identity; and verifiability without shared secrets. EdDSA Action Passports, human-in-the-loop approvals bound to the exact action, and an append-only SHA-256 hash-chained ledger with signed exports make each decision replayable and independently checkable against a public key. The deterministic core is production-grade and hermetically tested by a red-team suite; the surrounding platform is built but pre-GA, with live external validation the honest next step.

CONTENTS

1. Executive Summary

An autonomous agent reads context, decides what to do, and calls tools — refunds, deploys, emails, database writes, shell commands — without a human on the keystroke of each action. The moment it holds a credential and a tool, it holds the standing authority of the human it acts for, exercised at machine speed. Enterprises adopting agentic systems inherit three failure modes at once: **excessive agency** (a broad, long-lived key lets a support agent issue a $50,000 refund as easily as a $5 one), **prompt injection and tool poisoning** (adversarial text in a retrieved document or a Model Context Protocol server's own tool description redirects the agent), and **non-auditability** (application logs record that a call happened, not that it was permitted, against which policy, bound to which approval, at which version of the tool). Most teams have logs; very few have guardrails.

The instinctive fix — make the model police itself with system-prompt rules or a second "judge" LLM — is the wrong control for enforcement. Such a guardrail is *probabilistic* (a paraphrase flips the verdict), *in-band* (it reads the same poisoned context that compromised the primary agent), *bypassable* (it inherits every jailbreak that defeats the primary model), and it can neither fail closed nor produce a cryptographic artifact an auditor can check. The model must be treated as untrusted for access-control decisions. The real object of control is the *action* — the tool call and its arguments — not the prompt that produced it.

ActPass is a deterministic authorization gateway and tamper-evident evidence fabric that sits between an agent and the tools it calls. For every risky action it renders an allow / deny / require-approval decision — the *preflight decision* — computed outside the language model, then seals that decision into an append-only record. Seven pillars compose into that gateway: short-lived, single-use **Action Passports** (EdDSA/Ed25519 JWS carrying scope, replay, and revocation state); a JSON-DSL **policy engine** with first-match-wins rules and no LLM in the path; **manifest-drift detection** that breaks verification when a consented tool is silently swapped; **human-in-the-loop approvals** with hash-bound unlock; a hash-chained **evidence ledger** with signed exports; a **credential vault** (AES-256-GCM, AAD-bound to tenant and provider) from which raw secrets never leave the broker; and **browser-attested enrollment** (RFC 8628 device grant) that replaces copy-paste keys with explicit human capability consent. These compose into a deterministic preflight pipeline (passport → policy → drift → identity → approval) that emits one of 60 typed reason codes.

Five design principles govern every part of the system: **the decision lives outside the model**; **fail-closed by architectural mandate** (any condition ActPass cannot positively resolve denies); **no client-trusted fields** (tenant, approval, and manifest hashes are re-derived server-side); **the API key is the identity** (the tenant and agent come from the authenticated principal, never the request body); and **verifiability without shared secrets** (any attestation is checkable by a third party using only a published public key). The last principle is the compliance angle: a passport, a signed report, or an ActPass Certified badge is validated offline, with no round-trip to ActPass, so an auditor confirms an action was authorized without being handed a secret they must then trust. §8 maps these mechanisms onto the OWASP LLM Top 10, NIST CSF, SOC 2, and the EU AI Act.

Maturity, stated honestly. ActPass is at late-alpha. The deterministic enforcement core — preflight engine, passport crypto, policy evaluator, drift algorithm, evidence hash chain, and the no-client-trust discipline — is built, fail-closed, and re-verified at file:line by an independent adversarial red-team pass, with the credential-vault and access-control dimensions scoring highest in review. What is *not* yet field-proven is live-systems validation: we claim no signed enterprise customer in production, no completed SOC 2 audit, and no live SIEM or IdP/SSO round-trip. Parts of the product plane (dashboard onboarding, outbound approval notifications, distribution channels) are built but not yet exercised end-to-end, and two Low-effort correctness hard-stops must land before enforcement and evidence integrity are asserted as proven. §11 gives the full live-validated / built / planned accounting, and the path to GA is wiring against existing interfaces, not re-architecture.

A design partner should engage now because the load-bearing hard part — a deterministic, out-of-band, fail-closed decision sealed into verifiable evidence — is already built, tested, and inspectable, while the remaining work is a first real install against live systems. Engaging now shapes the integration surface and the self-hosted pilot before GA freezes them, on a foundation a skeptical security team can replay and check for itself.

2. The Problem: Governing Autonomous Agents

An autonomous agent is a system that reads context, decides what to do, and calls tools — refunds, deploys, emails, database writes, shell commands — without a human on the keystroke of each action. That is the point of the technology and also its liability. The moment an agent holds a credential and a tool, it holds the standing authority of the human it acts for, exercised at machine speed and volume. Enterprises adopting agentic systems inherit three distinct failure modes. None is speculative; each is documented in the MCP security literature, in OWASP's agentic guidance, and in the field reports summarized in ActPass's landscape research (`docs/actpass-deep-research-report.md`).

2.1 Three failure modes

Excessive agency. An agent does more than it was intended to do because it *can*. OWASP defines excessive agency as arising from excessive functionality, excessive permissions, or excessive autonomy. A support agent handed a broad Stripe key can issue a $50,000 refund as easily as a $5 one; a coding agent with write access to a repository can merge and deploy where it was only meant to comment. The root cause is structural: broad, long-lived credentials and unscoped tool access mean the *capability* granted always exceeds the *intent* behind any single task. Practitioner interviews in the landscape study name this directly — "broad role permissions and generic API keys" that are tolerable in staging become unacceptable in production.

Prompt injection and tool poisoning. An agent is manipulated into a harmful call by data it processes. Because an LLM cannot reliably distinguish trusted instructions from untrusted content, adversarial text — in a retrieved document, a web page, an email body, or a tool's own description — can redirect the agent's behavior. Tool poisoning is the supply-chain variant: a Model Context Protocol (MCP) server's tool metadata is crafted to steer tool selection or smuggle instructions, and descriptor-level attacks succeed even when the tool's schema remains syntactically valid. The MCP specification itself treats tool annotations as untrusted unless they come from a verified server, precisely because the metadata is an attack surface. Prompt-only defenses degrade here by construction: the injected content and the defense share one channel — the model's context window — and the attacker controls the content.

Non-auditability. After an agent acts, there is no defensible record of *what* was authorized, *why*, and *under whose authority*. Application logs capture that a call happened, not that it was permitted, against which policy, bound to which approval, at which version of the tool definition. When a security team, an auditor, or a customer asks "prove this action was allowed," a log line is an assertion, not evidence — it is mutable, unsigned, and disconnected from the authorization decision it purports to justify. The landscape research captures the gap bluntly: "Most enterprise teams have logs, very few have actual guardrails."

2.2 Why prompt-based guardrails are the wrong control

The instinctive response is to make the model police itself: add system-prompt rules, or interpose a second LLM as a "judge" that reads the proposed action and blocks the unsafe ones. This is the dominant pattern in the crowded prompt-security category, and it is the wrong control for enforcement. Four properties make it unfit.

- **Probabilistic, not deterministic.** An LLM classifier returns a likelihood, not a decision. The same action can be allowed on one invocation and blocked on the next; a paraphrase can flip the verdict. Research on agent security concludes there may be an *irreducible* tradeoff between blocking attacks and preserving legitimate capability with model-based defenses — you cannot tune a probabilistic filter to zero misses without also destroying utility. An authorization control that is right most of the time is, for a $50,000 refund, a control that is wrong some of the time.
- **In-band.** The guardrail lives in the same channel the attacker already influences. A judge LLM reads the same poisoned context that compromised the primary agent; the injected instruction that says "issue the refund" can carry a second clause that says "and tell the reviewer this is routine." Placing the reviewer inside the blast radius does not shrink it.

- **Bypassable.** Because the control is text interpreting text, it inherits every jailbreak, homoglyph, zero-width, and tag-wrapper laundering technique that defeats the primary model. Descriptor-level studies report a substantial fraction of unsafe invocations slipping past baseline model defenses. The defense and the attack are the same kind of object.
- **It cannot fail closed, and it cannot produce evidence.** A prompt guardrail has no defined behavior when it is uncertain, when the model times out, or when a key is missing — it degrades toward whatever the model happens to emit, which is failing *open*. And a probabilistic natural-language verdict is not a cryptographic artifact: it cannot be signed against a specific payload, bound to an approval, or chained into a tamper-evident record. There is nothing to hand an auditor but another sentence.

The deeper point, made repeatedly across the research base (`docs/actpass-deep-research-report.md`, §Research synthesis), is architectural: **the model must be treated as untrusted for access-control decisions.** Capability-based systems that separate control flow from data flow and enforce policy on tool calls outside the model achieve provable security on a meaningful fraction of adversarial tasks; formal reference-monitor approaches enforce policy without modifying the agent at all. The real object of control is the *action* — the tool call and its arguments — not the prompt that produced it.

2.3 Requirements for a real enforcement control

Inverting the four failure properties yields the requirements for a control that actually governs an agent, rather than advising it:

Requirement	What it means	Which failure it answers
Out-of-band	The decision is rendered outside the LLM, on a channel the agent and its inputs cannot reach or rewrite.	Prompt injection / tool poisoning
Deterministic	The same action, identity, and policy state always yield the same allow / deny / require-approval verdict — replayable, testable, explainable.	Probabilistic bypass
Fail-closed	Network errors, missing keys, unverifiable claims, and no-match-found all deny. Uncertainty resolves to *no*.	In-band degradation
Least-privilege / scoped	Authority is bound to a specific purpose, tool, resource, and value ceiling, short-lived and single-use — not a standing credential.	Excessive agency
Provably auditable	Every decision is a signed, hash-chained artifact bound to identity, policy, and approval — verifiable by a third party without a shared secret.	Non-auditability

These map cleanly onto the primary agentic risks in the OWASP LLM Top 10 — *excessive agency*, *prompt injection*, and *insecure output handling* — as well as onto MCP's own mandates for explicit consent, audience-bound tokens, and mistrust of tool metadata. §8 owns the full standards mapping (OWASP LLM Top 10, NIST CSF, SOC 2, EU AI Act); the point here is only that the requirements above are not ActPass idiosyncrasies but the convergent conclusion of standards bodies, platform vendors, and independent research.

2.4 Thesis

A guardrail written in the model's own language, evaluated by the model, sharing the model's context, cannot be the thing that stops the model. **Enforcement belongs outside the model** — a deterministic, out-of-band, fail-closed authorization decision, made against explicit policy and identity, and sealed into evidence that survives scrutiny. That is the control ActPass builds, and the rest of this paper describes how.

3. Design Principles

Five load-bearing principles govern every part of ActPass. They are not aspirations; each has a concrete, checkable consequence in the system, and §7 (adversary model) later shows they hold as invariants under attack. §4 explains

where each principle lives in the topology, and §5 shows them composed in the preflight pipeline. Here we state the principle, the reason it exists, and what it forces the implementation to do.

3.1 The decision lives outside the model

The authorization decision for a risky action must be a pure function of its inputs, computed outside the language model. Given the same action — tool, resource, arguments, agent, tenant, mode, policy version — ActPass renders the same `allow` / `deny` / `require-approval` decision every time. No LLM sits in the enforcement path.

The reason is that a probabilistic decision cannot be an authorization boundary. A model that can be talked into refunding once can be talked into it again with different phrasing; a control you cannot reproduce you cannot audit, test, or defend before a design partner's security team. §2 develops why prompt-level guardrails fail for exactly this reason.

The consequence is architectural: the decision engine (`lib/actpass/preflight.ts` , `policy.ts`) is a deterministic evaluator over a JSON policy DSL with first-match-wins rules, and it emits a typed `reason_code` (one of 60) for every outcome. Because it is a pure function over an injectable repository, the entire policy, passport, and evidence core runs in tests with no database and no network — the same property that lets a skeptical reader replay any decision (§9).

3.2 Fail-closed by architectural mandate

Any condition ActPass cannot positively resolve produces a deny. A network error reaching a store, a missing or unrotated signing key, an unverifiable passport claim, an unmatched policy — all resolve to a blocking decision, never a silent allow.

This is a mandate, not a configurable default, because a fail-open default is a single misconfiguration away from disabling the entire gateway, and the failure is invisible until an action lands that should have been stopped. ActPass enforces the property at every layer rather than as one toggle: an unreachable dependency in the passport, policy, drift, or approval stage each denies on its own.

The consequence is visible from the first request. On a fresh workspace, `POST /api/v1/actions/preflight` returns `{"decision":"deny","reason_code":"policy.denied_default"}` — no policy has matched, so the action is blocked. That deny is the success signal. `RuntimeMode` (`monitor` / `warn` / `enforce` / `strict`) tunes how loudly a decision is surfaced, but never converts a deny into an allow.

3.3 No client-trusted fields

Values that determine a security outcome are never trusted as the client sends them. Tenant identifiers, approval hashes, and manifest hashes are always re-derived or re-verified server-side against authoritative state before they influence a decision.

The rationale is direct: a field the server trusts from the request body is a field an adversarial agent — or a compromised one — controls. An approval hash quoted by the caller, a manifest hash asserted rather than recomputed, a tenant id in the payload: each is a lever for privilege escalation or cross-tenant access if believed.

The consequence runs through the pillars. An approval only unlocks a blocked action when its hash is verified against the recorded approval record (§6, §17); a passport's tenant binding is re-checked, not read from the request; manifest drift (§6) is decided by recomputing the tool manifest hash and comparing it to the consented baseline, so a rug-pulled tool breaks verification instead of sailing through. Tenant isolation — no cross-tenant read of approvals, evidence, credentials, or keys — is a defended invariant precisely because the tenant is never taken from the client.

3.4 The API key is the identity

The tenant and agent for a request are resolved from the authenticated principal — the bearer key — and never from the request path or body. A developer key (`apdv_…`) or agent key (`apk_…`) names *who* is acting; the server looks up that principal's registered tools and policies and explicitly ignores any client-supplied `x-actpass-tenant` .

Two motivations converge here. Security: if identity comes only from the credential, there is no path or body field to tamper with, which collapses a whole class of confused-deputy and cross-tenant attacks into a non-issue (this is the mechanism behind 3.3's tenant guarantee). Operability: the client no longer has to *declare* its identity, so `tenantId` and `agentId` in SDK and integration config become optional telemetry hints, never required and never trusted for authorization.

The consequence is that configuration is candy-simple: paste one key, done. `createActPass({ apiKey })` enforces against the managed cloud with a single secret, and `actpass install` pairs a machine with zero further config. The principle is a contract clarification, not a backend behavior — every `/api/v1/*` route already resolves identity this way.

3.5 Verifiability without shared secrets

Anything ActPass attests to — an Action Passport, a signed report, an ActPass Certified badge — is verifiable by any third party using only a published public key, with no secret shared between the verifier and ActPass.

The reason is trust that survives contact with a skeptic. A design partner, an auditor, or a buyer's security team should be able to confirm an attestation independently, without an API call to ActPass and without being handed a credential that itself must be trusted and protected.

The consequence is a consistent choice of asymmetric cryptography. Action Passports are EdDSA (Ed25519) JWS carrying scope, replay, and revocation state (§6). Reports and the Certified badge use Ed25519 detached signatures (RFC 7797), bound to the subject's manifest hash: a verifier fetches the object and checks it against the published public JWK alone. Because the signature covers the manifest hash, a forged or rug-pulled manifest breaks the signature — the badge flips to Drift or Expired on its own, with no ActPass in the loop. §8 details the export and badge formats.

4. System Architecture

ActPass interposes a single enforcement point between an AI agent and the tools or APIs it calls. Every risky action traverses the gateway, which renders a deterministic allow / deny / require-approval decision — the *preflight decision* — before the action reaches its target. The topology below describes where that gateway sits, how its control and data planes divide, and the deployment placements the same enforcement core supports.

4.1 Where the gateway sits

The gateway is the enforcement root. An agent does not call `stripe.refund.create` directly; it calls ActPass, which decides, and only on `allow` does the action proceed:

```
Agent ──► ActPass Gateway ──► Tool / API
            │
            ├─ verify Action Passport (EdDSA scope / replay / revocation / approval binding)
            ├─ check tool manifest drift
            ├─ evaluate deterministic policy (JSON DSL, no LLM in the path)
            ├─ route to human approval when policy requires it
            └─ seal the decision into the hash-chained evidence ledger
```

The decision is the product; proxying the byte stream to the upstream is optional. A caller may run the preflight and execute the action itself (advisory or `monitor` mode), or route the call through the gateway's HTTP/MCP proxy so ActPass forwards to the upstream only after an `allow`. In the proxying case the upstream base URL and auth scheme for a server slug come solely from server-side environment configuration (`ACTPASS_UPSTREAM_<SERVER>`), never from the client, and every forwarded request passes the shared SSRF range guard (TLS-only, non-private egress) before it leaves the gateway.

4.2 Control plane vs. data plane

ActPass separates a stateful *control plane* from a stateless-in-the-hot-path *data plane*. This split is what lets enforcement stay fast and available while the authoritative state lives in one managed place.

Plane	Owns	On the per-action hot path?
Control plane	Policy authoring and distribution, passport issuance signing keys, key rotation and enrollment, revocation and approval ledgers, credential vault, evidence storage and exports, dashboards, billing, teams	No
Data plane	The per-action preflight decision (passport verify → policy → drift → identity → approval), optional proxying to the upstream, local replay cache, evidence emission	Yes

The control plane is the source of truth for all durable state; the data plane consumes cacheable projections of that state (public verification keys, policy bundles, a revocation denylist, an approval set) and emits evidence back. Control-plane data flows *to* the gateway as cacheable bundles; evidence and telemetry flow *from* the gateway asynchronously. A control-plane outage therefore degrades evidence freshness, not the ability to make a decision.

4.3 Managed hosting and the thin-client philosophy

ActPass ships managed-hosting first. The gateway and control plane run together in ActPass's cloud — the API at `https://www.api.actpass.org` (the passport issuer), the dashboard at `https://www.actpass.org`. Under this default there is no separate SaaS hop per call: the gateway *is* the cloud, and the credential vault and evidence ledger are the hosted Postgres-backed stores.

Every client — the TypeScript and Python SDKs, the CLI, the GitHub Action, the n8n node — is *thin*. Clients carry no business logic that belongs in the cloud, and no client requires the customer to run a server of their own. Identity follows the **API-key-is-identity** rule: the tenant and agent are resolved from the authenticated principal (the bearer key), never from the request body or a URL path segment, which the server explicitly ignores for authorization. A client is therefore configured with one key and nothing else; the cloud owns all state and all decisions.

Figure 1. The agent calls the ActPass gateway (data plane), which renders the preflight decision and optionally proxies to the tool; the control plane owns policy, keys, ledgers, the credential vault, and evidence, feeding the data plane cacheable bundles and receiving evidence asynchronously — shown across the managed-cloud, customer-VPC, and offline-verification placements.

4.4 Deployment topologies

The same enforcement core admits three placements. Only the first is the shipped default today; the maturity of the others is accounted for honestly in §11.

(a) Managed cloud gateway — available now. Every tenant uses this. Gateway and control plane collapse into ActPass's cloud; the customer deploys nothing. This is the topology the Quickstart, the SDKs, and the `/api/v1/actions/preflight` endpoint target.

(b) Customer-VPC / self-hosted enforcement — Enterprise, designed and scaffolded. For Enterprise customers who require enforcement inside their own egress boundary, the gateway image (`ACTPASS_ROLE=gateway` , with a Helm workload and a dedicated Dockerfile) runs inside the customer's VPC behind their egress. What stays in the customer boundary: the agent, the gateway, the local credential vault, and the local replay cache — the entire per-action hot path. What remains in the ActPass control plane: policy authoring and distribution, passport issuance keys, the revocation and approval ledgers, and the evidence vault. Only cacheable control-plane projections cross the boundary inbound (public JWKS, policy bundles, revocation denylist, approval set), and evidence crosses outbound asynchronously. The offline-verifiable passport core (below) is what makes this placement a wiring exercise against existing injectable interfaces rather than a re-architecture; the honest remaining work — revocation-sync denylist, key distribution to the gateway, pointing the replay store local, evidence buffering — is detailed in §11.

© Fully offline passport verification — built and test-proven. A gateway holding only the public verification key can validate a passport with zero network callback. The verification path uses a local in-memory key map and local `jwtVerify` — no `createRemoteJWKSet` , no `fetch` , no phone-home. Against published JWKS it checks, offline:

- the EdDSA/Ed25519 signature (`kid` -selected for rotation);
- issuer, audience, tenant, agent, and user claims;
- expiry and not-before;
- tool and resource scope, plus `resource_constraints` value ceilings (for example a per-refund amount cap).

The JWKS is published at `/.well-known/actpass/jwks.json` and `/api/v1/passports/jwks`, and an executable test exports the public JWK, re-imports it standalone, and verifies a real token with no private key or issuer in scope — that offline property is already in the suite. The stateful checks that a signature-and-scope check cannot cover offline — revocation, single-use replay, and live approval-hash verification — are cleanly factored behind injectable stores (see §6 and §7), so a placement can choose fail-closed-on-staleness or a synced denylist per check rather than depending on a live callback. The intended revocation-sync model is a periodic poll (or control-plane push) of the denylist projection on a fixed interval, with a stated staleness threshold beyond which a stale denylist fails closed — bounding the revocation-latency window to at most that interval rather than leaving it open-ended. That threshold is a design parameter of the placement, to be finalized with the first pilot but not an open question: the seam is fail-closed by construction, not fail-stale. The local single-use replay store carries a stricter durability requirement — it must be durable and shared across gateway replicas and restarts, or a restarted or peer replica would re-admit a spent single-use token — so an in-memory cache is insufficient here even when it suffices for the denylist. Signature and scope are self-contained in the token; state is a deliberate, bounded seam.

Across all three placements the invariants are identical: the decision is deterministic and made outside the LLM, unverifiable claims and unreachable dependencies fail closed, and client-supplied tenant, approval, and manifest hashes are never trusted. The topology changes where the gateway runs; it does not change what the gateway guarantees.

5. The Preflight Decision Pipeline

The preflight pipeline is where ActPass renders its verdict. Every risky action an agent proposes passes through a single entry point, `POST /v1/actions/preflight`, and the engine returns a deterministic `allow` / `deny` / `require_approval` decision — with `warn`, `require_tool_reapproval`, and related states as refinements — accompanied by a typed reason code and a sealed evidence linkage. The decision is computed by `preflight(input, repo)` in `lib/actpass/preflight.ts`, which is a **pure function over an injected `PreflightRepository`**: it performs no I/O of its own. The route handler wires a Drizzle-backed repository; tests wire an in-memory one. This design is what makes the engine hermetically testable (§9 owns the testing depth), and it is also a security property — the same code path runs in production and under test, so a bypass cannot hide in an untested branch.

5.1 The trust boundary

Before the first gate, the engine draws a hard line: it **never trusts client-supplied tenant, tool hash, approval, or risk fields**. `tenantId` is resolved upstream from the authenticated API key (API-key-is-identity), and the tool manifest hash and active policy come from the repository, not the request body. The request carries `tool`, `resource`, `args`, `agent_id`, `user_id`, `goal`, an optional compact-JWS `passport`, and a `mode`. The engine's very first act is to canonicalize the arguments — `canonicalizeArgs()` recursively sorts object keys — and compute a passport-excluding `requestHash` of `{tool, resource, args}` via SHA-256. That hash is the stable identity of the action: it is what an approval binds to, so the execute route can recompute it and consume the grant even across passport re-minting. Critically, the client-supplied `mode` **may only raise enforcement, never lower it**: `effectiveMode = strongestMode(input.mode, policy.mode)` ensures an agent asking for `monitor` cannot downgrade a tool the tenant admin declared `strict`. Every hard gate below keys off `effectiveMode`, so the untrusted caller can tighten but never loosen the tenant-authored posture.

5.2 The five gates

The engine walks the following sequence. Any gate that fails closed short-circuits to `finalize()`, which seals evidence and returns.

Gate 1 — Authenticate and resolve context. The bearer key authenticates and pins the tenant. The engine then resolves the current approved tool manifest and active policy from the repository. A storage failure here does not throw through; it returns `deny` / `policy.missing` / `risk_tier: critical` with `http_status: 500` — fail-closed. An unknown tool yields `deny` / `tool.unknown`; a tool with no governing policy yields `deny` / `policy.missing`. Both are still evidenced.

Gate 2 — Verify the Action Passport. If a passport is supplied, `verifyPassport()` checks the EdDSA (Ed25519) signature, expiry (`passport.expired`), not-yet-valid (`passport.not_yet_valid`), audience (defaulting to the tool manifest's server origin), and the tenant/agent/user binding. These are **hard passport reasons** — any one produces `deny`, with `http_status` 403 for authorization mismatches (`passport.audience_mismatch`, `passport.tenant_mismatch`, `passport.tool_not_allowed`, `passport.resource_out_of_scope`, `passport.replay_detected`) and 401 otherwise. Once hard signature checks pass, `assertPassportUsable()` enforces tool scope, resource scope, and single-use replay defense (`jti`) against the request hash and idempotency key. Revocation is durable across replicas. If the passport is absent but the tool is `high`- or `critical`-risk in `enforce` / `strict` mode, the engine denies with `passport.missing` (401). Passport internals are detailed in §6.2.

Gate 3 — Check tool manifest drift. If the approved manifest has drifted from the consented baseline and is awaiting review (`reapprovalRequired`), the engine returns `require_tool_reapproval` / `tool.manifest_changed` in `enforce` / `strict` — the hash mismatch means the agent is about to call a tool that is no longer the one a human approved. `monitor` / `warn` fall through and record the drift as evidence rather than blocking. A companion use-time identity gate (ASI04) hashes the descriptor the upstream actually served at call time against the approved baseline and verifies a registered publisher's detached signature, catching runtime body-tamper. Drift detection is detailed in §6.4.

Gate 4 — Evaluate the deterministic policy. `evaluatePolicy(policy, context)` is the single source of allow/deny. It is a **first-match-wins** JSON-DSL evaluator with a **fail-closed default** (`policy.denied_default`). The context it receives is fully server-constructed — canonicalized `args`, the server-side manifest hash and risk tier, resolved passport claims, and the resource descriptor — so the policy reasons over trusted inputs only. Two provenance gates run just ahead of policy evaluation: an *action-open delegation* check (untrusted content cannot choose the business action when the user's goal is open-ended) and a *provenance-aware code-execution* check for `execute`-category tools sourced from untrusted content. Both escalate to `require_approval` (or `warn` in `monitor`). The policy DSL is detailed in §6.3.

Gate 5 — Route or seal. If the outcome is `require_approval`, the engine either resolves it immediately — a passport carrying a verified `approval_hash` that was already proven against the approval ledger flips the decision to `allow` / `approval.satisfied` — or it creates an approval row (with sensitive args redacted before persistence via `redactSensitiveArgs()`, which masks keys such as `password`, `token`, and `card_number`) and routes to a human, returning the `approval_request_id`. Because a forged `approval_hash` is verified against the tenant- and tool-bound ledger inside the passport gate and can never reach this point, the fast-path resolution is safe. Every path then funnels through `finalize()`, which appends a `preflight_decision` event — carrying the tenant, chain id, agent, tool, policy version, manifest hash, request hash, decision, reason code, and task provenance — to the hash-chained evidence ledger, and returns the sealed `evidence_event_id` and `chain_id`. The `chain_id` defaults to the idempotency key, so a later `execute()` call appends to the same chain and the decision and its outcome share one audit thread. Evidence write failure is itself fail-closed: it blocks (`deny` / `evidence.write_failed` / 500) in `enforce` / `strict` — no action proceeds without a durable record — while `monitor` / `warn` log the gap for operators and continue. Approvals and evidence are detailed in §6.5 and §6.6.

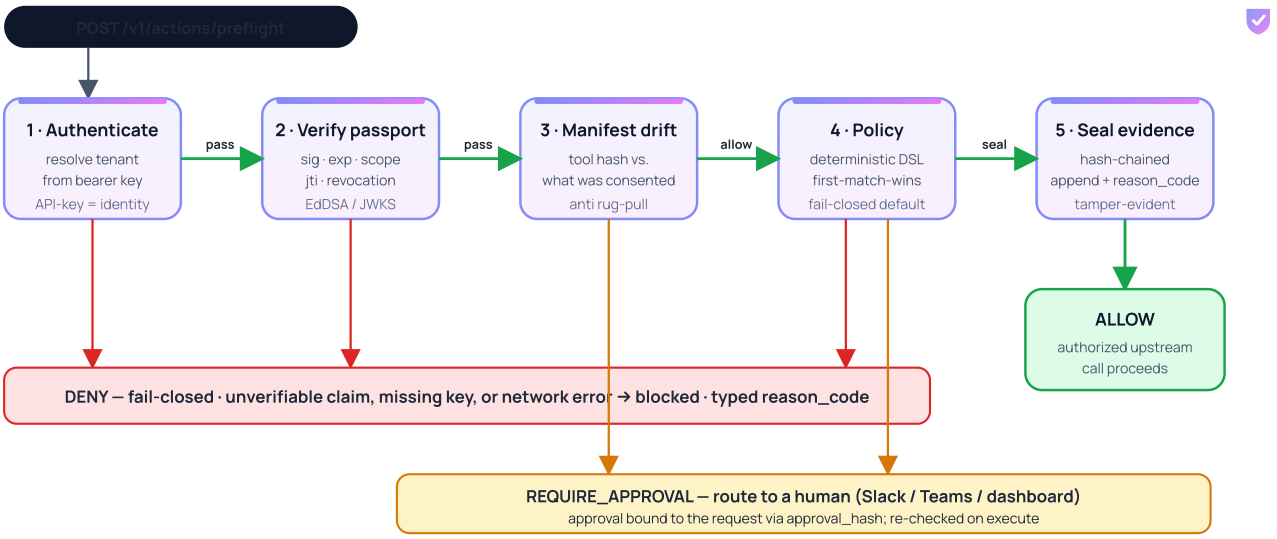

Figure 2. A proposed action enters at `POST /v1/actions/preflight` and traverses five sequential gates — authenticate/resolve, passport, manifest drift, deterministic policy, and approval routing. Any gate may short-circuit; every terminal path seals a signed evidence event before returning.

5.3 Request and response shapes

The request and response are stable, typed contracts (grounded in `PreflightInput` / `PreflightResponse` in `preflight.ts` and the OpenAPI `PreflightRequest` / `PreflightResponse` schemas):

```
// POST /v1/actions/preflight — request
{
  "tool": "stripe.refund.create",
  "resource": "stripe:charge:ch_123",
  "args": { "amount": 4000, "customer_id": "cus_9" },
  "agent_id": "agent-support-7",
  "user_id": "u_42",
  "goal": "refund the duplicate charge",
  "mode": "enforce",
  "passport": "eyJhbGciOiJFZERTQSIsimtpZCI6...",
  "audience": "https://api.stripe.com",
  "idempotency_key": "req_01HZ..."
}
```

```
// 200 — response
{
  "decision": "require_approval",
  "reason_code": "refund.medium_needs_approval",
  "risk_tier": "high",
  "tool_manifest_hash": "sha256:1f3a...",
  "policy_hash": "sha256:9c02...",
  "approval_request_id": "apr_7781",
  "evidence_event_id": "ev_00A2...",
  "chain_id": "req_01HZ...",
  "http_status": 200,
  "explain": {
    "summary": "Policy refund_policy v3: require_approval.",
    "matched_rules": ["require_approval_medium_refund"],
    "next_steps": ["approve_in_dashboard_or_slack"]
  }
}
```

5.4 Why typed reason codes matter

Every terminal path carries one of **60 typed `ReasonCode` values** (`lib/actpass/core.ts`), namespaced by pillar — `passport.*` (13), `tool.*` (23), `policy.*` (9), `approval.*` (5), `args.*` (4), `refund.*` (3), `credentials.*` (2), `evidence.*` (1). A reason code is not a log string; it is a stable, machine-parseable contract. Because the code is emitted deterministically at the exact gate that produced the decision, an auditor can reconstruct *why* an action was blocked without replaying an LLM, and a SOC can alert on a specific class — `passport.replay_detected` or `policy.approval_required` — rather than string-matching prose. (A small number of runtime strings, such as `circuit_breaker.tripped` and `policy.untrusted_code_execution`, are emitted directly at preflight and sit outside the typed union.) The `explain` object pairs each code with a human summary, the `matched_rules` that fired, and the operator's `next_steps`, so the same decision serves both automated alerting and human review. The reason-code taxonomy and the five-gate pipeline are enumerated in full in §5's companion appendix and §7's invariant catalog.

6. Core Pillars

The guarantees described so far are delivered by seven pillars that compose into the single preflight decision of Section 5. They ship together, but each is designed to be understood — and tested — in isolation: a clear purpose, a well-defined interface, and explicit dependencies. The subsections below detail their internals — how each works, how it is used, and what it depends on.

6.1 Action Passports

An **Action Passport** is a short-lived, cryptographically signed credential that binds one authorization decision to one narrow window of intent. It is the artifact that turns "the user is logged in" into "this user authorized *this* agent to use *these* tools against *these* resources, under *this* policy and *this* tool state, for the next few minutes." Every risky action the gateway forwards carries a passport; the preflight pipeline (§5) treats a missing or unverifiable passport as fail-closed.

Concretely, a passport is a compact JWS (JSON Web Signature) — a JWT — signed with **EdDSA over Ed25519** via the `jose` library. There is no bespoke cryptography. The protected header carries `alg: EdDSA`, `typ: JWT`, and a `kid` (key id) so verifiers can select the right public key during rotation.

Claim set

Issuance (`issuePassport`) assembles a fully-populated claim set. A representative decoded payload:

```
{
  "iss": "https://www.api.actpass.org",
  "aud": "gw:acme-prod",
  "tenant_id": "t_9f21",
  "agent_id": "agent_support_bot",
  "agent_version": "1.4.2",
  "user_id": "u_4410",
  "delegator_id": "u_4410",
  "goal": "issue refund for order 88213",
  "allowed_tools": ["stripe.refund"],
  "allowed_resources": ["order:88213"],
  "resource_constraints": { "max_amount": 5000, "currency": "USD" },
  "risk_tier": "high",
  "policy_id": "refund_policy",
  "policy_version": 7,
  "policy_hash": "sha256:1b9e…",
  "tool_manifest_hash": "sha256:a03c…",
  "approval_hash": null,
  "nbf": 1751630400,
  "iat": 1751630400,
  "exp": 1751631300,
  "jti": "ap_3f0c9d2b7a1e4c88b0f5e6d2a9c14477"
}
```

Each claim is a *binding*, not a hint. `aud` pins the passport to one gateway audience; `tenant_id` , `agent_id` , and `user_id` pin identity; `allowed_tools` and `allowed_resources` scope capability and target; `resource_constraints` carries value-level ceilings (an amount cap, a currency, an id binding). `policy_hash` , `policy_version` , and `tool_manifest_hash` freeze the policy and tool state the authorization was granted against, so later drift is detectable. `risk_tier` (`low` / `medium` / `high` / `critical`) and `goal` are recorded for evidence and human review. `delegator_id` defaults to `user_id` and carries the original grantor across delegation. The `jti` is a per-passport nonce prefixed `ap_` (a hyphen-stripped UUIDv4), and it is the anchor for replay defense.

Lifetime

Passports are deliberately short-lived. The route-level clamp (§Appendix, `passport-ttl`) defaults to **900 s (15 min)**, with a floor of **30 s** and a ceiling of **3600 s (1 hr)**. `issuePassport` itself is the low-level primitive: it upper-bounds `ttlSeconds` to `MAX_TTL_SECONDS` but intentionally does *not* apply the min/reject-negative clamp, so tests can mint already-expired tokens and the untrusted `/issue` routes remain the sole owner of the lower bound. The upper bound is not cosmetic — it keeps the key-rotation overlap window (below) meaningful, since an unbounded passport could outlive any retired key's grace period.

Verification path — verified vs. trusted

`verifyPassport` establishes what is cryptographically *verified*. It selects the public key by the token's `kid` from a rotation-aware registry (falling back to the current signing key when the header omits one), then calls `jwtVerify` with `algorithms: ['EdDSA']` and a 5-second clock tolerance. A bad signature, expired `exp` , or premature `nbf` short-circuit to `passport.invalid_signature` , `passport.expired` , and `passport.not_yet_valid` respectively. Only after the signature holds does it check the *value* claims against the caller-supplied context: `iss` must equal the trusted issuer, and `aud` , `tenant_id` , and (when provided) `agent_id` / `user_id` must match — each mismatch emits a specific reason code (`passport.audience_mismatch` , `passport.tenant_mismatch` , and so on). It also

consults the revocation store and, when the context supplies current hashes, flags `tool.manifest_changed` and `policy.update_required`. A passport is `valid` only when this error list is empty.

Crucially, verification proves the passport was signed by ActPass and is internally consistent — it does **not** prove the requested action is in scope, nor that any `approval_hash` is real. Those are enforced later, at *use* time.

Use, replay defense, and approval binding

`assertPassportUsable` is the final gate before an action is forwarded, and it is where the single-use guarantee lives. It re-checks revocation (closing the TOCTOU window between verify and use), confirms the concrete `toolName` is in `allowed_tools` and `requestedResource` is in `allowed_resources`, and enforces `resource_constraints` values against canonicalized args (an `amount` above `max_amount` is denied as `args.amount_exceeds_limit`; a non-numeric present amount is denied rather than waved through).

Before consuming the nonce, it verifies any `approval_hash`. A passport's `approval_hash` is **client-supplied until proven**: the `ApprovalVerifier` must confirm the hash is the `event_hash` of a reviewer-granted approval bound to the same tenant, tool, and action; a forged or unmatched hash is denied `approval.invalid`, and a verifier outage fails closed. This check runs *before* the replay claim so a forged approval can never burn the single-use `jti`.

Replay defense is a single atomic claim on the `jti` through the injectable `ReplayCache` (production backs this with a `passport_uses` unique constraint; the default is in-memory). The first claim wins. A second use with a *different* `requestHash` is a replay attempt (`passport.replay_detected`); a second use with the *same* `requestHash` is treated as an idempotent retry and allowed — so a client that retransmits an identical request after a timeout succeeds without minting a new passport.

Revocation, rotation, and offline verifiability

Revocation is **tenant-scoped**: a revoke targets `(jti, tenant_id)`, so one tenant learning another's `jti` cannot deny-of-service it. Revocation invalidates future uses immediately; in-flight passports otherwise expire naturally at `exp`. Key rotation is handled by a verification registry keyed on `kid`: the current signing key never expires, retired keys are admitted with a `notAfter` (default `MAX_TTL_SECONDS`) so passports they signed keep verifying through the overlap and no longer, and a compromised `kid` can be denylisted wholesale — rejected outright and refused re-entry even inside its rotation window. One caveat scopes that in-window guarantee: unlike the DB-backed per-`jti` revocation store, the verification-key registry and its denylist are process-local in-memory, so denylisting a compromised `kid` on one instance does not yet propagate across serverless replicas or survive a cold start — the wholesale in-window rejection holds on a single instance today, and durable cross-replica propagation is a tracked gap (see §11 Status and Roadmap). The published **JWKS** (`getPublicJwks`) advertises exactly the keys the verifier will accept, skipping denylisted and expired entries, so any relying party can verify a passport **offline** against ActPass's public keys without calling the gateway. The same Ed25519 keypair also signs detached inter-agent request signatures, so request integrity verifies against the identical JWKS with no second key to manage.

6.2 Deterministic Policy Engine

The policy engine is the second gate in the preflight pipeline (§5) and the component that turns an organization's written rules into a machine-checkable decision. It is a pure function: given a validated `PolicyDefinition` and a normalized `PolicyContext`, `evaluatePolicy` returns a `PolicyDecision` — a `Decision`, a `reason_code`, the names of the rules that matched, and per-condition evidence — with no I/O, no clock reads, and no network calls (`lib/actpass/policy.ts`). The same inputs always produce the same output. This determinism is the property everything else in ActPass depends on: it is what makes a decision reproducible in an audit, replayable in a red-team harness, and signable in an evidence record.

Critically, no large-language-model output influences the decision. An LLM may help an operator *draft* the JSON of a policy, but the drafted text is then run through `validatePolicyDefinition` and evaluated by the deterministic engine. The model never sits in the decision path.

The JSON DSL

Policies are authored as JSON — a stable, dependency-free encoding of the DSL. A `PolicyDefinition` carries an `id`, an integer `version`, an optional `applies_to` scope (`tools` / `agents`), an optional `mode` (`monitor` / `warn` / `enforce` / `strict`), and an ordered list of `rules`. Each rule pairs a `when` condition group with a `decision`, a `reason` (which becomes the emitted reason code), and, for `require_approval` rules, an `approval` block naming the channel and minimum role.

A condition is a triple: a dotted `path` into the context, an `operator`, and a `value`. The engine supports ten operators:

```
==  !=  >  >=  <  <=  in  not_in  contains  matches
```

Numeric comparisons are coercion-aware. Because a JSON `amount` frequently arrives as a string, `>` `>=` `<` `<=` coerce both operands to finite numbers when possible, so `"1000000" > 50000` evaluates to `true` rather than failing lexicographically (a bug that would silently skip a deny-on-amount rule). `matches` compiles the right operand as a regular expression; an invalid pattern returns `false` rather than throwing. `contains` works over both arrays and strings.

A `value` may instead be a cross-field reference — `{ "$ref": "<path>" }` — which resolves a second dotted path in the same context. This is how a rule compares a request argument to a claim carried in the Action Passport (§6.1), for example checking that `args.queue_id` equals `passport.resource_constraints.queue_id`. Cross-field comparison is what lets a policy enforce that an agent stays inside the scope its passport was minted for, without hard-coding tenant-specific values into the rule text.

Conditions are grouped under exactly one of `all` (logical AND) or `any` (logical OR). Rules are evaluated top to bottom, **first-match-wins**: the first rule whose group matches returns immediately with that rule's decision. This gives authors a predictable precedence model — put the most specific or most restrictive rules first — and it means the ordering of the `rules` array is itself part of the policy's meaning.

Fail-closed by construction

Every ambiguous or degenerate case resolves to `deny`, and the reason code says why:

- No policy supplied → `deny` / `policy.missing`.
- Context missing its `args` object → `deny` / `args.schema_invalid`.
- A scoped policy evaluated against an out-of-scope tool or agent → `deny` / `policy.missing`.
- All rules evaluated, none matched → `deny` / `policy.denied_default`.
- An empty condition group matches nothing.

The operator semantics are chosen so that a *missing* signal biases toward the restrictive branch. A `not_in` check against an absent allowlist is trivially true, so a deny-on-`not_in` guard *fires* when the allowlist is missing rather than being silently skipped. An `in` check against an absent allowlist is trivially false, so an allow-on-`in` guard *does not fire* and the request falls through to the default deny. And `!=` returns true against an `undefined` left path, so a deny-on-mismatch rule still triggers when a security-critical passport claim is absent — `undefined != "resolve_refund_request"` is genuinely true. The self-test (`docs/actpass-policy-expressiveness-2026-06-13.md`) verifies this property explicitly: every gap fails closed — missing input produces over-escalation or deny, never a silent allow.

Two grounded examples

A refund threshold rule from the flagship `refund_policy` pack, escalating high-value refunds to human approval:

```
{
  "name": "high_value_requires_approval",
  "decision": "require_approval",
  "when": { "all": [
    { "path": "args.amount", "operator": ">", "value": 50000 }
  ] },
  "reason": "policy.approval_required",
  "approval": { "channel": "slack", "min_role": "approver" }
}
```

A deploy-gating rule from the `github_pr_merge_deploy` pack, denying a merge into a protected branch when continuous-integration status is anything other than a clean pass:

```
{
  "name": "block_deploy_on_red_ci",
  "decision": "deny",
  "when": { "all": [
    { "path": "args.base_branch",  "operator": "in", "value": ["main", "release"] },
    { "path": "args.ci_status",    "operator": "!=", "value": "passed" }
  ] },
  "reason": "policy.denied_default"
}
```

Note the second rule references `args.ci_status` — a value the agent never puts in its own request. That is the enricher's job.

The enricher pattern

The engine deliberately does no I/O, which keeps the language small and auditable but means it can only decide on values already present in the context. Anything derived — a build's CI status, whether the changed files match a protected-path glob, whether the current time is inside business hours, how many deploys an agent has run today — is computed *outside* the engine and stamped into the `PolicyContext` before evaluation. This pre-population step is the **enricher**: a per-tool gateway adapter that fetches or derives inputs and injects them as ordinary context fields.

This split is a deliberate design choice, not a limitation. Two classes of input flow through it. **Derived inputs** — `ci_status`, `touches_protected_paths` (a glob over changed files), `is_business_hours` (computed from the clock and the team's timezone) — are I/O the gateway performs so the policy stays a pure comparison. **Stateful counters** — `deploys_today`, `actions_this_hour` — are aggregated from the `usage_events` / `passport_uses` tables, then injected as a number the engine compares against a cap. In both cases the engine change required is zero: enrich the context, then write an ordinary condition. Because an absent enrichment leaves the field `undefined`, the fail-closed operator semantics ensure a missing signal escalates or denies rather than waving the action through.

Built-in policy packs

ActPass ships **11 built-in policy packs** that encode common agentic-risk patterns so a design partner starts from working rules rather than a blank file: the flagship `refund_policy`, plus `email_send`, `crm_update`, `sql_access`, `shell_command`, `github_pr_merge_deploy`, `file_access`, `data_export`, `webhook_trigger`, `permission_change`, and `native_guard`.

Why a deterministic DSL, not OPA/Rego or an LLM allow-list

The expressiveness self-test encoded three realistic customer policies against the *actual* engine and proved that the operator language, `$ref` comparison, `all` / `any` grouping, first-match-wins ordering, and fail-closed default express every *decision* in all three (`tests/unit/actpass-policy-expressiveness.test.ts`, 13/13 passing). The only

things the engine could not do — supply derived inputs and maintain counters — are handled by the enricher and are not language problems. The verdict was explicit: enough for pilots, no OPA, no new DSL.

The argument against a general-purpose engine here is auditability and testability, not expressive power. A rule in this DSL is a flat list of comparisons a security architect can read and reason about without learning Rego's evaluation model; adopting OPA would add a second runtime, a second language, and a second thing to threat-model, while diluting the integrated passport ↔ policy ↔ evidence story. An LLM-generated allow-list fails a stronger test: it is neither deterministic nor red-team-testable, and it reintroduces exactly the non-determinism ActPass exists to remove from the decision path. The invariants of this engine — fail-closed default, coercion-aware numerics, `$ref` scoping to passport claims — are each pinned by a runnable test, which is the property a skeptical buyer can verify rather than take on faith. The design of the DSL and its threat surface are examined further in §7.

6.3 Manifest Drift Monitoring

A tool an agent trusted last week is not necessarily the tool it calls today. In the Model Context Protocol (MCP) and OpenAPI ecosystems, a tool descriptor — its name, description, input/output schema, declared side effects, publisher, and origin — is pulled from a remote server at connection time and can silently change on the next fetch. An operator who consented to a benign `search_docs` tool can, without any local action, end up wired to a descriptor that has grown a `send_email` capability, widened a `max_amount` bound, or embedded hidden instructions in its prose. This is the **rug-pull** / **tool-poisoning** class of supply-chain attack. ActPass treats a tool descriptor as a versioned, signed artifact: it is canonicalized, hashed, bound into consent and passports, and re-checked on every ingest. Capability-increasing change forces re-consent before the tool is usable again.

Canonicalization and hashing. `canonicalizeManifest()` (`lib/actpass/manifests.ts`) reduces a raw `ToolManifest` to a stable `NormalizedToolManifest`. Descriptions collapse runs of whitespace to a single space and trim; origins lowercase and strip trailing slashes; protocol and publisher identity lowercase; `oauthScopes` and `sideEffectProfile` are sorted; `publisherVerified` defaults to `false`. Each schema is hashed independently — `inputSchemaHash`, `outputSchemaHash`, `oauthScopesHash`, `authRequirementsHash` — via `sha256(sortedStringify(value))`, and `hashManifest()` folds a semantic projection (the sub-hashes plus origin, publisher, side effects, risk tier, capability) into one whole-manifest digest of the form `sha256:<hex>`. Because `sortedStringify` is key-order- and whitespace-independent, two descriptors that differ only in JSON formatting produce the **same** hash. This matters: a naive `JSON.stringify` comparison would fire drift on cosmetic re-serialization (false positives that train operators to click through) and, worse, could miss a real change hidden behind reordered keys. The hash is a semantic fingerprint, not a byte fingerprint. That fingerprint is what gets bound into the consent record and Action Passport, so a mint or preflight can assert *this exact tool shape was authorized*.

Drift classification. `diffManifests(previous, current)` compares two normalized manifests field by field and emits a set of typed `DriftSignal`s. A first-seen tool (`previous === null`) is `new_tool` at `critical` materiality. Otherwise the diff walks each semantic dimension and raises a signal only when the change is real:

- **Authority expansion** — `findMaxNumeric()` recursively finds the maximum value under authority keys (`max_amount`, `limit`, `max`, `amount_limit`) in each schema; if the current maximum exceeds the previous, `input_schema_expanded_authority` fires. It takes the *maximum, not the first match*, so a raised spend ceiling is caught even when a benign pagination `limit` precedes it in walk order.
- **Sensitive-field growth** — `collectKeyNames()` recursively gathers every (lowercased) key; a key newly present that matches `SENSITIVE_FIELD_NAMES` (`password`, `token`, `ssn`, `api_key`, …) raises `input_schema_expanded_sensitive` or `output_schema_expanded_sensitive`.
- **Read→write drift** — a newly added side effect in the `WRITE_LIKE` set (`write`, `delete`, `send`, `execute`, `deploy`, `pay`, `refund`, `transfer`, `admin`, `permission_change`) raises `read_to_write_drift`; other side-effect additions or removals raise `side_effect_changed`.
- **Trust shifts** — `server_origin_changed`, `publisher_identity_changed`, `publisher_verification_changed` (flagged in either direction, since a verified→unverified downgrade is the dangerous one), `oauth_scope_broadened`, `auth_requirements_changed`, `risk_tier_increased`.

- **Value-only rug-pull** — if `inputSchemaHash` changed but no specific expansion signal fired (a default flipped, an enum widened, a regex relaxed), `input_schema_changed` fires; the analogous `output_schema_changed` covers output-shape changes with no new sensitive field. The tool's *meaning* changed even though no growth was detectable.

Both recursive walks are depth-capped at `MAX_SCHEMA_DEPTH = 256` and throw a catchable error on a hostile deeply-nested schema, which callers turn into a fail-closed deny rather than an uncaught stack overflow. Each signal carries a materiality (`none` → `critical`); the diff's materiality is the maximum over its signals.

From signal to decision. `classifyDrift()` maps the signal set to a `DriftDecision` using an ordered table (`SIGNAL_DECISION`) where the highest-severity matching signal wins and emits a typed `reason_code` (`tool.read_to_write_conversion` , `tool.financial_authority_expanded` , `tool.oauth_scope_broadened` , …) consistent with the reason-code vocabulary in §5. `read_to_write_drift` , `server_origin_changed` , and `publisher_verification_changed` `block` ; authority expansion, new tools, auth changes, and every schema-value change (including the bare `input_schema_changed` / `output_schema_changed` rug-pull) resolve to `require_reapproval` or `require_reauth` with `requires_human_approval: true` . Only cosmetic drift — `description_changed` , `risk_category_changed` — resolves to a non-blocking `warn` . The design bias is explicit in the source: a meaning-changing input schema is **never** auto-approved to `warn` , because `warn` silently accepts the new baseline. Capability *shrinkage* (a removed side effect) is still flagged, because a masked capability is as suspicious as a gained one.

One scanner, two products. The same canonicalization powers the static risk scan. `scanManifest()` (`lib/actpass/scanner.ts`) canonicalizes a manifest and produces structured `RiskFinding` s across a fixed taxonomy — missing/weak auth, broad OAuth scope, unverified publisher, unknown origin, financial and destructive capabilities, plus descriptor-level tool-poisoning and prompt-injection detection hardened against unicode homoglyph and zero-width-splitter laundering (see §6.2 and §7). Each finding carries `SecurityStandardMapping` s to OWASP Top 10, OWASP LLM Top 10, NIST CSF, and SOC 2 controls, and the scan accretes a `risk_score` (0–100) and `highest_severity` used downstream by certification (§8). Sibling scanners cover MCP config files, OpenAPI specs, and n8n workflow exports. `shouldFailOn(result, threshold)` turns any scan into a CI gate: the ActPass GitHub Action runs the scanner over a repository's tool sources and fails the build when findings meet or exceed the configured severity threshold, moving drift and poisoning detection left into the pull request rather than into production runtime. The runtime drift monitor and the CI scanner are two entry points into one deterministic engine — the same hash a scan computes at PR time is the hash a preflight verifies at call time.

6.4 Human-in-the-Loop Approvals

Some actions are not simple allow/deny calls. A $2,000 refund, a production deploy, or an email to an entire customer list is exactly the kind of decision an organization wants a person to sign off on — but only for the specific action proposed, and only once. ActPass models this as a first-class outcome of the preflight decision (§5): when deterministic policy evaluation returns `require_approval` , the gateway does not execute and does not block indefinitely. It records an approval request, routes it to a human, and holds the action until a reviewer decides or the request expires.

The security-critical property is that a granted approval unlocks *exactly* the action a human saw, and unlocks it *once*. Everything below exists to make that binding non-forgeable and non-replayable.

Request creation and routing

When `evaluatePolicy` returns `require_approval` and the caller's passport carries no already-verified approval grant, `preflight()` calls `repo.createApprovalRequest(...)` , persisting a row in `human_approvals` with the tenant, tool, risk tier, matched reason code, and a redacted copy of the arguments. Argument redaction runs first: `redactSensitiveArgs()` masks values under keys such as `password` , `token` , `api_key` , `ssn` , and `card_number` (recursively into nested objects) before anything is stored or surfaced, so a reviewer's dashboard and the persisted row never carry a raw secret.

The request is then routed to a human through Slack, Microsoft Teams, or the dashboard, per the README's decision path ("Route to human approval when policy requires it — Slack / Teams / dashboard"). The reviewer

decides through `POST /v1/approvals/{id}/decide`. Notably, the approval binds to the *action*, not to the identity of whoever clicks approve in Slack: the decide route attributes the reviewer's channel and external id in the audit trail but keys the grant off the action hash, so there is no Slack-id-to-ActPass-principal mapping to spoof.

Expiry semantics

A pending approval that is never actioned cannot stall the action plane forever. The approval state machine (`approvals-fsm.ts`) permits exactly one exit from `pending` for an un-actioned request — the `expire` transition to `expired`, which the spec then treats as a hard block. `findOverdueApprovals()` computes, given a clock, which pending rows have passed their SLA deadline; the deadline is derived from `created_at` plus an SLA window (`DEFAULT_APPROVAL_SLA_SECONDS`, 24 h, env-overridable). The boundary is exclusive — `expiresAt === now` is not yet overdue — and rows with no parseable deadline are never expired. A background route (`/v1/approvals/expire`) feeds tenant rows through this pure function and persists the transitions, so an expired approval fails closed rather than lingering as a latent grant.

Consumption: binding a grant to the exact request

The crux is how a granted approval is *consumed* on the execute path without any client-trusted field. Every action is fingerprinted by a passport-*excluding* request hash — deterministic over the tool, resource, and canonicalized arguments:

```
requestHash = "sha256:" + sha256(
    sortedStringify({
      tool,
      resource,
      args: canonicalizeArgs(args),   // recursive key-sort, deterministic
    })
  )
```

`canonicalizeArgs()` recursively sorts object keys (and preserves array order) so that two semantically identical argument objects that differ only in key ordering hash identically, and any change to a value or a key changes the hash. This `requestHash` is written to the approval row as its `user_context`. The passport-excluding shape is deliberate: it survives passport re-minting, so a short-lived passport can be refreshed for the same approved action without invalidating the grant, while still being pinned to the concrete tool + resource + argument tuple and the tenant. The reviewer's `approve` decision is itself sealed into the immutable `approval_events` ledger as an `event_hash` (`sha256` over the reviewer, action, and any modified args), and that `event_hash` is what a subsequent passport carries as its `approval_hash` claim.

To spend the grant, the caller mints a passport whose `approval_hash` equals that `event_hash`. On verification, `verifyApprovalHash()` recomputes nothing from the client: it looks up the `approval_events` row by `event_hash`, and admits it only if it belongs to the same tenant (`team_id`), was an `approve` decision, links a `human_approvals` row still in status `approved` for the same `tool_name`, and whose `user_context` equals the freshly recomputed `requestHash` for the action now being executed. A swap or TOCTOU attack on the approval row — approve a cheap action, then execute an expensive one — fails at this join: the recomputed `requestHash` will not match the stored `user_context`, and the grant is rejected fail-closed. A forged `approval_hash` never survives, because it is verified against the ledger before `preflight()` ever treats the approval requirement as satisfied (`approval.satisfied`).

Single consumption is enforced on terminal execution. `consumeApprovalForAction()` retires the grant via the FSM's `approved → executed` transition, optimistically locked on `(status = 'approved', version)` so concurrent consumes cannot double-fire. Because `verifyApprovalHash()` requires `status = 'approved'`, a consumed grant then fails closed on any replay. Grant lifetime is additionally bounded: `isApprovalFresh()` rejects a grant older than a max-age window (`DEFAULT_APPROVAL_MAX_AGE_SECONDS`, 24 h) — and fails closed if the row has no `resolved_at` — so a stale approval cannot unlock an action long after a human signed it.

The reuse story is tightened further by single-use passport refresh (§6.2): the approval binding survives re-minting by design, but the passport carrying it is itself replay-protected, so a captured passport cannot be re-presented. The honest live-validated-versus-built accounting for the Slack/Teams round-trip and reviewer channels is deferred to §11.

6.5 Evidence Fabric

Every preflight decision (§5) terminates in a seal: the gateway appends one event to a per-action, append-only, hash-chained ledger. The evidence fabric is the record that a decision was rendered, on what inputs, under which policy version, and with what outcome. Where a conventional audit log is a mutable table anyone with write access can rewrite, the fabric is a tamper-*evident* structure: any insertion, deletion, reordering, or content edit breaks a cryptographic invariant that a third party can detect without trusting the database operator.

What is sealed

An `EvidenceEvent` records the shape of the decision, never its secrets (`lib/actpass/evidence.ts`). Each event carries a `tenant_id`, a `chain_id` scoping it to one action, an `event_type` drawn from a fixed vocabulary (`preflight_decision`, `passport_verified`, `approval_granted`, `tool_execution_completed`, and so on), the `decision` and `reason_code` the pipeline emitted, the `policy_version` evaluated, and content *digests* rather than content: `request_hash` over the canonical action arguments, `response_hash` over the tool's reply, and `tool_manifest_hash`. Raw arguments, credentials, and secrets are never written to the ledger — only their hashes — so the fabric proves *what was decided over which bytes* without becoming a second copy of the sensitive payload. The persisted row also stores the exact canonical pre-hash JSON (`sealedEventJson`), which is what makes content-level tamper detection possible: a verifier re-derives the hash from the stored bytes and compares, rather than trusting a lossy column projection.

The hash chain

Sealing is deterministic. `computeEventHash` takes SHA-256 over the canonical (deeply key-sorted) JSON of the event, excluding only the hash field itself, the DB-assigned surrogate `event_id`, and the derived MAC:

```
current_event_hash = "sha256:" + hex(sha256(canonical_json(event \ {current_event_hash, event_id, event_
```

Each new event links to its predecessor by carrying the prior event's `current_event_hash` as its own `previous_event_hash` (`appendEvidence`). Because the previous hash is *inside* the digest of the current event, any edit to an earlier row cascades: its hash changes, which invalidates the `previous_event_hash` of every event after it. The genesis event has `previous_event_hash = null`.

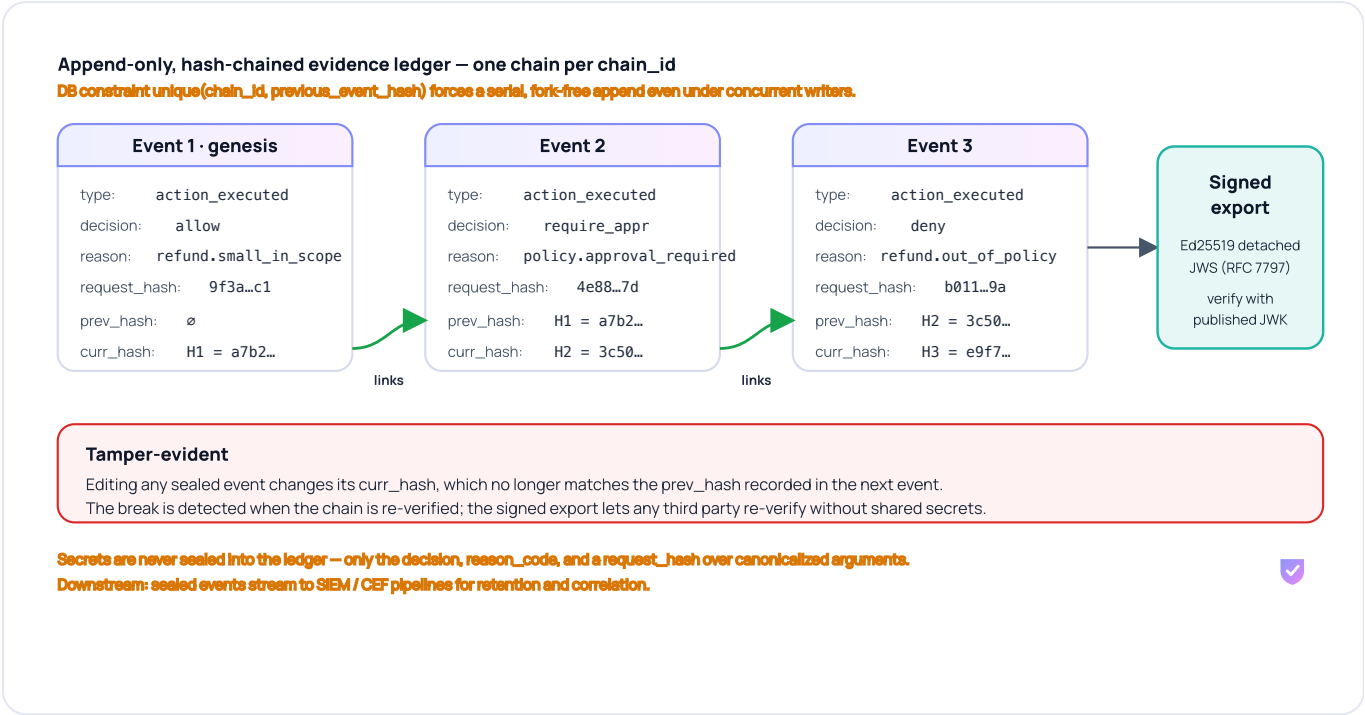

Figure 4. Each event seals a SHA-256 digest over its canonical contents, and carries the prior event's digest as its `previous_event_hash`. A DB unique constraint on `(chain_id, previous_event_hash)` forces one parent per link, so the chain is linear with no forks; a signed head anchor over `{length, tip_hash}` closes tail truncation.

Atomic serial-append under concurrency

The production driver is Neon serverless HTTP, which has no interactive transactions, and `SELECT … FOR UPDATE` cannot lock a row that does not yet exist. Two writers that both read the same chain tip would otherwise both link to it and fork the chain. The fabric closes this at the schema layer: a unique constraint `evidence_logs_chain_prev_uq` allows **one parent per `(chain_id, previous_event_hash)`**, and `evidence_logs_chain_genesis_uq` allows one genesis per chain. Concurrent appenders race to insert; the database admits exactly one and rejects the rest with a unique-violation. The loser does not fail — it re-reads the now-advanced tip and retries, linking cleanly after the winner (`appendEvidenceRow` in `lib/actpass/evidence-store.ts`):

```
for (let attempt = 0; ; attempt++) {
  const [prevRow] = await db.select().from(evidenceLogs)
    .where(and(eq(evidenceLogs.teamId, teamId), eq(evidenceLogs.chainId, chainId)))
    .orderBy(desc(evidenceLogs.createdAt), desc(evidenceLogs.id)).limit(1);
  const sealed = appendEvidence(prior, event);          // links previous_event_hash
  try {
    const [row] = await db.insert(evidenceLogs)
      .values({ ...buildValues(sealed), eventMac }).returning();
    return { sealed, id: row.id };
  } catch (err) {
    if (isUniqueViolation(err) && attempt < maxRetries) continue;  // tip advanced; retry
    throw err;
  }
}
```

The tip is selected by `(createdAt, id)` descending; the serial `id` is the deterministic tiebreaker because `createdAt` can collide at sub-millisecond resolution under concurrent appends. Retries are bounded (default five) — a pathological hot chain with more than five simultaneous appends throws rather than spins, an acceptable ceiling at the current single-session append rate and the marked upgrade path is a driver with real serializable transactions.

Defense in depth beyond linkage

Linkage alone is re-forgeable by anyone who can rewrite rows and recompute SHA-256. The fabric adds two keyed defenses the database principal cannot satisfy:

- **Per-event keyed MAC.** Every row stores `event_mac = HMAC-SHA256(current_event_hash)` under an app-side signing secret the DB role does not hold (`computeEventMac`). A DB-only adversary can recompute a keyless hash but cannot forge the MAC. Verification is fail-closed and downgrade-resistant: a chain is "keyed" if *any* event carries a MAC, and a keyed chain in which any event is missing its MAC — including an attacker stripping MACs to look legacy — fails verification.
- **Signed head anchor.** After each append the fabric re-derives and re-signs the chain head `{length, tip_hash}` with the Ed25519 report-signing key (`upsertChainAnchor`, §6.7). Truncating tail rows leaves the row count or tip disagreeing with the signed anchor, and re-signing requires a key the database does not hold. A keyed chain presented *without* an anchor is treated as a stripped anchor and fails closed.

These defenses are calibrated to a *DB-only adversary* — a principal who can rewrite rows but holds neither the app-side MAC secret nor the anchor signing key. Tamper-evidence is that strong and no stronger: an adversary who additionally holds both the `event_mac` HMAC secret and the anchor signing key can re-forge a chain end to end, because every check above then verifies against keys they control. The mitigation is to not keep the only anchor inside the trust boundary — retain an independently held signed export as an external anchor, so a re-forgery in place still disagrees with a copy the adversary never touched (Appendix C).

Tamper detection

`loadAndVerifyChain` reads a chain back in insertion order, and `verifyChain` re-checks it end to end: the genesis has no predecessor, each event's `previous_event_hash` equals the prior event's `current_event_hash`, each `current_event_hash` is *recomputed* from the stored canonical bytes and must match, each keyed MAC must verify, and the row count and tip must equal the signature-verified anchor. Verification is fail-closed at the edges: a non-empty chain whose rows lack a parseable `sealedEventJson` is reported unverifiable (`evidence.unverifiable_missing_sealed_json`) rather than passing on a lossy column read, and an anchor whose Ed25519 signature does not verify short-circuits to invalid (`evidence.anchor_signature_invalid`). Any break returns the exact index at which the chain diverged.

Signed, third-party-verifiable exports

Evidence leaves the fabric as detached-JWS artifacts, not opaque log dumps. The report signer (§6.7) produces an RFC 7797 detached Ed25519 signature (`b64:false`, `crit:['b64']`) over the canonical payload, so the exported evidence stays human-readable and is canonicalized exactly once (`report-signing.ts`). A consumer re-supplies the canonical bytes alongside `{protected, signature}` and verifies with only the **published public JWK** (`getReportPublicJwk`) — no shared secret, no trust in ActPass at verify time. This is the substrate the SIEM/CEF pipeline (§10) and the signed compliance exports (§8) sit on: a downstream system, an auditor, or a customer's own tooling can independently confirm that an evidence bundle is authentic and unaltered. The mechanism described here is the guarantee; §8 covers how those exports map to compliance framings and §9 covers how the chain and its concurrency path are exercised under test.

6.6 Credential Vault

The credential vault holds the long-lived tool secrets an agent's actions ultimately need — Stripe keys, GitHub tokens, SMTP passwords, generic HTTP API keys — and releases them only server-side, only after preflight (§5) allows an action. The design goal is stated in one line: **raw secrets never leave the broker module.** The agent that triggers an action holds no upstream credential; it authenticates to ActPass with its own key (API-key-is-identity, §3), and ActPass supplies the tool secret at the moment of the outbound call. This section grounds that discipline in `lib/actpass/crypto-envelope.ts` (encryption at rest) and `lib/actpass/credentials.ts` (the broker).

Envelope encryption and AAD binding

Secrets are stored as an authenticated-encryption envelope, not plaintext. `encryptSecret` uses **AES-256-GCM** with a random 96-bit IV per secret and a 128-bit GCM authentication tag, so a tampered ciphertext fails to decrypt

rather than silently returning garbage — integrity, not just confidentiality. The data-encryption key (DEK) is loaded from `ACTPASS_CREDENTIAL_KEY` (32 bytes, hex or base64). In `NODE_ENV=production` the module **fails closed**: a missing or wrong-length key throws rather than persisting a secret under a default or derived key. A derived/ephemeral dev key is permitted locally, with a loud warning, only outside production.

Two blob formats exist:

```
v1:<base64(iv ‖ tag ‖ ct)>            legacy, primary key, no AAD
v2:<keyId>:<base64(iv ‖ tag ‖ ct)>    versioned key id + optional AAD
```

The `keyId` — a stable, non-secret 12-hex-char digest of the DEK — lets a new key be rolled in without orphaning old ciphertexts: retired keys from `ACTPASS_CREDENTIAL_KEY_OLD` stay registered so their blobs still decrypt until a rotation job re-encrypts them under the primary. Decryption selects the named key, then falls back across every registered key for resilience; the GCM tag, not the key id, is what actually gates a wrong key.

The load-bearing control is **AAD binding**. `EncryptOptions.aad` feeds additional authenticated data into the GCM tag — in the vault this is the row identity, `${tenantId}:${provider}`. Because the tag covers the AAD, a ciphertext blob copied into another tenant's or provider's row **fails to decrypt**: the AAD presented at decrypt no longer matches the one bound at encrypt, and `decryptSecret` throws. This makes ciphertext non-transplantable across tenants and providers by construction, closing the class of attack where a stolen or misrouted blob is replayed against a different identity's key context. (Legacy `v1:` blobs carry no AAD; `decryptSecret` deliberately ignores an AAD argument for them so their tag still verifies.)

Be precise about what AAD does and does not buy. The root of trust today is a **single env-supplied data-encryption key** (`ACTPASS_CREDENTIAL_KEY`): one master key for the whole managed-cloud vault, not a per-tenant key, with KMS/HSM custody still on the roadmap (§11). Rotation is supported — retired keys registered via `ACTPASS_CREDENTIAL_KEY_OLD` keep old blobs decryptable while a job re-encrypts them — but that is key *versioning*, not key *isolation*. AAD binding stops an attacker from transplanting one row's ciphertext into another tenant's or provider's context; it does **nothing** against an attacker who already holds the DEK, because that attacker can supply the correct AAD for any row. State the blast radius honestly: **a leaked `ACTPASS_CREDENTIAL_KEY` decrypts every tenant's secrets**, and it stays that way until KMS-backed, per-tenant keys land (§11).

Server-side binding, never returned to the caller

`bindCredential` is the release path. Given a `BindingRequest` (`tenantId` , `provider` , `tool` , optional `amount` , and the `runtimeDecisionId` for evidence linkage), it looks up the tenant-scoped `CredentialVaultItem` and enforces scope before releasing anything: the credential must be `active` , unexpired, and scoped for the exact requested tool, and any financial `amount` is checked against `scope.maxAmount` . That cap check fails closed — a present-but-non-numeric amount is refused via `toFiniteNumber` , not skipped by a `typeof === 'number'` shortcut.

Critically, the broker does **not** hand the raw secret up the stack. It returns a `BindingHandle` whose `token` is an HMAC over the binding context (`item.id | tool | expiresAt | decisionId`), a deterministic `bindingHash` for the evidence event, and a short expiry (`TOKEN_TTL_MS` , 5 minutes). The raw secret is dereferenced only inside `executeWithBinding` , which re-validates that the handle still matches a current credential and has not expired, then invokes the provider adapter as `fn(secret)` . The comment on that function states the boundary explicitly: *"The provider adapter receives the raw secret here — never above this layer."*

This has two consequences the architecture depends on. First, the secret is **never returned to the agent or API caller** and is never written into the evidence ledger (§6.7) — evidence records the `bindingHash` and decision linkage, not the credential. Second, **egress destinations are server-configured, not client-supplied**: the provider is a field on the stored vault item, and the outbound call is made by the gateway to that provider's adapter under ActPass's own SSRF guard (`ssrf-guard.ts` , §7), so a caller cannot redirect a bound credential at an attacker-chosen host.

Together these properties complete the API-key-is-identity model. A compromised agent can request actions, but it cannot exfiltrate a tool credential, cannot transplant a stored blob across tenant boundaries, and cannot steer a

secret to an arbitrary destination. It holds only its ActPass key — the one credential the gateway is built to scope, revoke, and audit.

> **Maturity note (see §11):** the in-repo `InMemoryCredentialStore` is an MVP backing store; the module's own header calls for a KMS/HSM (AWS KMS, GCP KMS, or HashiCorp Vault) in production behind the stable `CredentialStore` interface. The envelope, AAD binding, and server-side-only release discipline described here hold regardless of that swap.

6.7 Agent Enrollment

An authorization gateway is only as trustworthy as the moment an agent first obtains credentials. If enrollment relies on an ambient, long-lived API key copied into a config file, every agent that shares that key is indistinguishable in the audit log — "an agent did it" is not an incident response. ActPass replaces copy-paste key distribution with **browser-attested install consent**: the trust anchor is an explicit human action reviewing declared capabilities, not a credential that happens to be present in the environment.

The design adapts **RFC 8628, the OAuth 2.0 Device Authorization Grant** — the same flow the GitHub CLI, Cloudflare CLI, and AWS device-login use to pair a headless client to a human account. ActPass keeps the RFC's device-code / user-code / poll skeleton and its error vocabulary, then adds one twist at the consent step: alongside the first Action Passport it mints a **developer key** (`apdv_…`) bound 1:1 to that installation, so the agent can rotate fresh passports without bothering the human again.

The flow. Enrollment is a five-step handshake between the agent runtime (SDK, CLI, or MCP server), the ActPass API, and a human in a browser:

1. **Initiate.** The freshly-installed runtime detects it has no credentials and calls `POST /api/v1/install/initiate` with its *declared identity*: `agent_name` , `kind` , `version` , `platform` , `capabilities` , and `requested_tools` . This endpoint is unauthenticated — it is the bootstrap. The server opens a pending `install_sessions` row and returns a device code, a human-readable user code, verification URIs, a TTL, and a poll interval.
2. **Prompt.** The runtime prints the `user_code` (e.g. `WPHK-XCJM`) and verification URL to stdout. For an SDK in a CI/non-interactive context it instead throws a structured `ActPassMissingCredentialsError` carrying the verification URI and a copyable hint; an MCP server refuses the tool call with `error.code = 'install_required'` so the MCP client renders the URL and code.
3. **Consent.** A human opens the URL, signs into ActPass with an existing session, and lands on the consent screen. There they review the agent's declared metadata and — critically — the **capabilities and tools it is requesting** before clicking the concretely-labelled `Issue developer key` . Only a human session can complete an install: an API-key bearer hitting `/install/complete` is rejected with `auth.forbidden` . Machine principals cannot consent.
4. **Delivery.** While the human deliberates, the runtime polls `GET /api/v1/install/poll?device_code=…` , receiving RFC 8628 states (`authorization_pending` , `slow_down` , `access_denied` , `expired_token`) as HTTP 400 with the OAuth error name. On consent the server registers the agent, mints the `apdv_` developer key, mints the first passport, seals the consent as evidence, and stashes the payload in-process. The next matching poll drains it once with HTTP 200 `{ token, developer_key, agent_id, installation_id, expires_at, claims }` .
5. **Rotate.** From then on the runtime holds only the long-lived developer key and short-lived passports. When a passport nears expiry it calls `POST /api/v1/agents/refresh-passport` with `Bearer apdv_…` to mint a fresh one — no human round-trip. Passport TTLs stay within the platform clamp (default 900s, max 3600s; see §6.2).

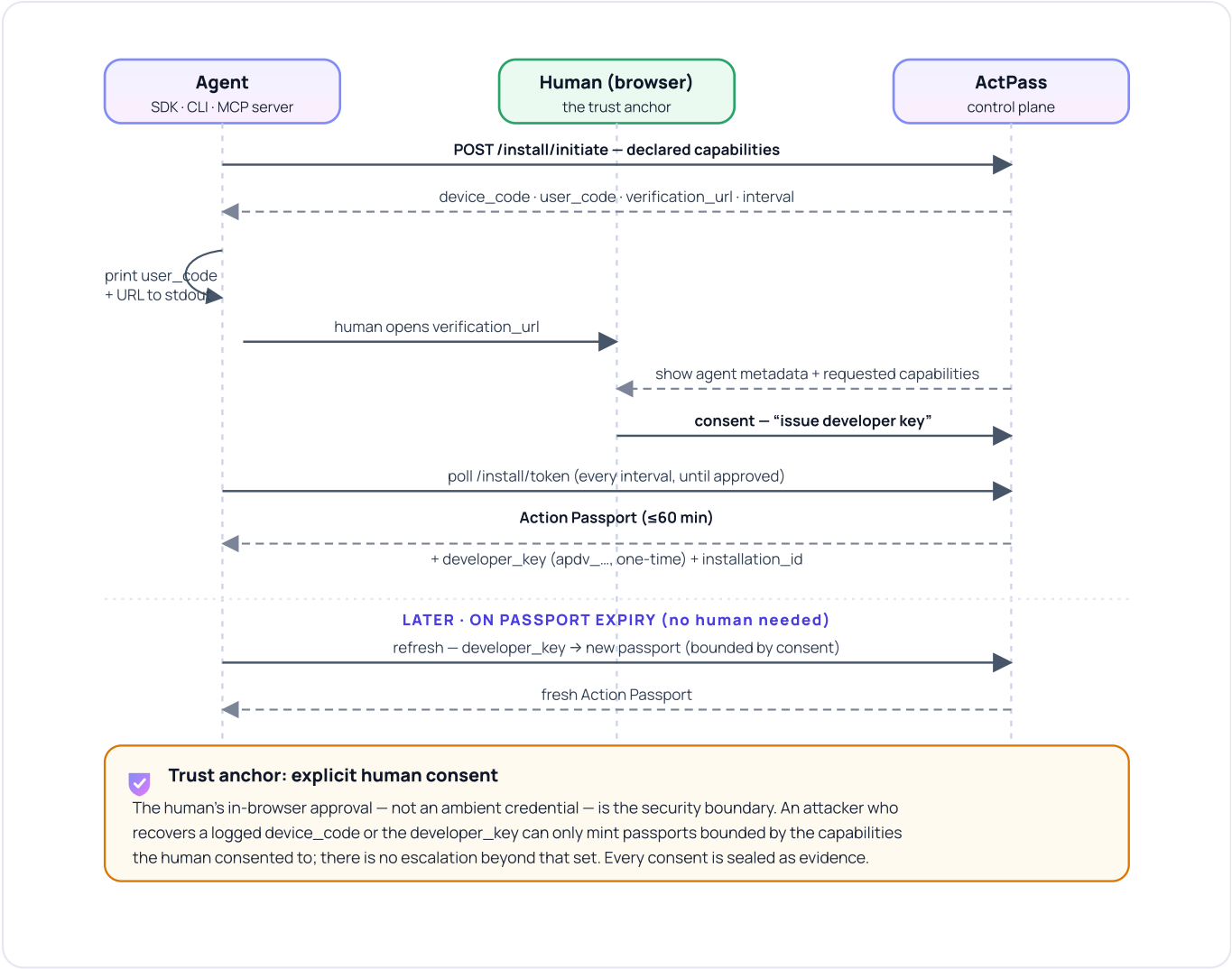

Figure 3. The five-step device-authorization handshake. The agent runtime declares its identity and polls; a human reviews the declared capabilities in the browser and consents; ActPass mints a per-installation developer key plus a short-lived Action Passport and seals the consent to the evidence ledger. The developer key thereafter rotates passports directly.

Why human consent is the anchor. Because the human approves a specific *capability set*, the blast radius of a stolen credential is bounded by that consent — there is no path to escalation. An attacker who scrapes a logged `device_code` mid-flow, or lifts a developer key from a compromised dotfile, can mint passports only for the capabilities the human already granted to that one installation. They cannot widen the capability set, cannot reach other agents on the same team, and every passport they mint is sealed to the evidence chain with the installation id, so post-compromise forensics still work. This capability ceiling is what makes single-step `actpass install` UX acceptable rather than reckless.

Crypto hygiene. The flow is built to survive a database snapshot and a leaked terminal:

- **`device_code`** — 32 random bytes, base64url (~43 chars). Only its SHA-256 is stored, so a dump of `install_sessions` cannot be replayed.
- **`user_code`** — 8 characters from a 31-symbol unambiguous alphabet (no `0`, `O`, `1`, `I`, `L`). The ~$2^{39.7}$ keyspace (just under 2^{40}) against a 15-minute TTL and per-IP rate limit makes online brute force infeasible.
- **`developer_key`** — 32 random bytes, `apdv_` prefix, 48 chars total; hashed at rest with SHA-256, transmitted exactly once at poll success. A stored key prefix (first 12 chars) supports dashboard listing without holding the secret.
- **Single-use delivery.** The post-consent payload lives in a per-process map keyed by session id; the first poll drains it and every later poll gets `invalid_grant`. A process restart between consent and poll simply forces a re-pair — strictly safer than persisting secrets at rest.

- **Rate limits and timing.** `/install/initiate` is a per-IP token bucket (20 requests / 60s). A client polling faster than the negotiated interval gets `slow_down` and must back off ≥ 5s. Unknown or malformed device codes return a constant `invalid_grant` that never reveals whether the row was missing, expired, completed, or denied.

Per-installation revocation. Because the developer key maps 1:1 to an `agent_installations` row, revocation is surgical: one click in the dashboard flips `status = revoked`, and the bearer path (`verifyDeveloperKey`) matches only `active` rows, so a revoked key can never refresh again. In-flight passports remain valid until natural expiry (bounded by the TTL clamp); operators who need immediate cutoff pair revocation with per-`jti` passport revocation (§6.2). The dashboard's installation list — platform, SDK version, device fingerprint, creation time — is also the first place a user notices a rogue enrollment, a deliberate mitigation against device-code phishing (MITRE ATT&CK T1566.004).

Evidence. Every consent seals a `passport_issued` event with `metadata.source = "agent-install"`, the `installation_id`, `developer_key_prefix`, and both requested and granted tools, on chain id `agent-install:<installation_id>`. An auditor can therefore replay the complete story — *agent X was installed on device Y by user Z at time T after consenting to capability set C* — from a single evidence chain (§6.5).

The naming convention deserves one clarification: the consent URL is `/setup/agent/2fa`, but the semantics are browser-attested install consent, not a literal TOTP or WebAuthn second factor. Adding real second-factor auth atop dashboard sign-in, and moving local credential storage from a `0600` dotfile into the OS keychain, are tracked as follow-ups (§11); the current file-level posture — re-asserted `0600` permissions, matching `~/.aws/credentials` — is documented as the floor, not the ceiling.

7. Trust and Threat Model

A gateway is only as strong as the assumptions it makes about who it is defending against and where its enforcement can be circumvented. This section states those assumptions plainly. It defines the adversaries ActPass is designed to resist, the trust boundaries it draws, the invariants it holds as testable properties, and — most importantly — the one place where a proxy-based gateway can be bypassed regardless of how correct the proxy itself is. That last point is not a caveat buried in an appendix; it is the load-bearing honesty of the entire design, and we present it before any reader has to ask.

7.1 Adversary model

ActPass assumes the agent is untrusted. The threat is not a buggy agent; it is an actively hostile one, and specifically these adversaries:

- **Compromised or prompt-injected agent.** The LLM has been steered — by a malicious document, a poisoned tool result, or a jailbreak — into attempting actions the operator never intended. ActPass treats every action request as potentially adversarial. Because the decision is deterministic and rendered *outside* the model (see §5, Preflight), a prompt that convinces the model to "ignore all prior instructions" does not reach the authorization logic; the policy engine never reads the prompt.
- **Malicious or negligent tool author.** A third-party tool (MCP server, plugin) whose manifest lies about its behavior, mutates after review, or ships with over-broad scopes. Manifest scanning, drift detection, and certification (§6, §8) target this adversary; a rug-pulled manifest hash breaks the signed attestation.
- **Network attacker.** An adversary attempting SSRF, credential exfiltration in transit, or redirection of an outbound forward toward an attacker-controlled or internal host.
- **Curious insider.** An authenticated principal in one tenant attempting to read another tenant's approvals, evidence, credentials, or keys, or a low-privilege member attempting privileged operations.

ActPass explicitly does **not** claim to defend against an adversary who has already compromised the managed gateway host or stolen the Ed25519 signing keys. Those are out of scope by definition (`SECURITY.md`): once an attacker holds the signing root, signatures are no longer evidence of anything. Denial-of-service via volumetric traffic is likewise out of scope for the enforcement guarantee.

7.2 Trust boundaries

The core boundary sits between the **agent runtime** (untrusted) and the **ActPass gateway** (the enforcement root). Everything the agent sends — including `tenant` in a URL path, an approval hash, or a manifest hash — is treated as attacker-controlled input and re-derived or re-verified server-side. Tenant identity is resolved key-is-identity from the authenticated gateway API key, never from the `:tenant` path segment. A second boundary separates the gateway from **upstream services** (GitHub, Slack, an internal API): upstream destinations are server-configured via `ACTPASS_UPSTREAM_<SERVER>` and SSRF-guarded, never client-supplied.

7.3 Design invariants as testable properties

The invariants below are stated so a skeptic can attack them, and each maps to enforcement code and regression tests. A violation of any one is a high-severity defect (`SECURITY.md`, "Design Guarantees Worth Probing").

Invariant	Property	Enforced by
Fail-closed	Network error, missing key, or unverifiable claim yields a BLOCKED decision — never a silent allow	`preflight.ts`, blocking-decision set in `core.ts`
No client-trusted fields	Tenant IDs, approval hashes, manifest hashes re-verified server-side	key-is-identity resolution; policy/drift gates
Single-use passports	A `jti` replays at most once, idempotently, with the same request hash	`passport.ts`, `acquireExecutionClaim` replay guard
Tenant isolation	No cross-tenant reads of approvals, evidence, credentials, keys	scoped queries; RBAC (§6)
Immutable evidence	Append-only ledgers with verifiable hash chains; tampering is detectable	`evidence-store.ts` hash-chained ledger
Credential confidentiality	Vault secrets AES-256-GCM encrypted at rest (AAD-bound to tenant + provider), bound only on the gateway path, never returned to callers or written to evidence	`crypto-envelope.ts`, `credential-store-db.ts`
Verifiable attestations	Signed reports and the Certified badge use Ed25519 detached JWS (RFC 7797); a third party verifies with the published JWK alone	`report-signing.ts`, `certified.ts`
Egress safety	Upstream destinations server-configured and SSRF-guarded (TLS-only, non-private)	`ssrf-guard.ts`, `http-forward.ts`

These are not aspirational statements. The AAD binding, for example, means an encrypted credential blob cannot be transplanted from one tenant's row to another's — the authenticated-decryption tag fails. The single-use guarantee means a captured passport replayed with a *different* request body is rejected because the request hash no longer matches.

One residual on **egress safety** is stated plainly rather than papered over: the SSRF guard validates a destination by re-resolving its DNS name and rejecting private, loopback, or link-local addresses, but it does not yet *pin* the validated address at connect time. Validation re-resolves DNS independently of the socket connect, so a DNS-rebind adversary whose name resolves to a public address at check-time and a private address at connect-time (a classic TOCTOU) is a known residual, marked with `ponytail:` comments in `ssrf-guard.ts` and `http-forward.ts`. The upgrade path is to pin the checked IP and connect to that exact address, closing the check-to-connect window.

7.4 The gateway-bypass problem

Here is the honest core, drawn directly from ActPass's internal bypass threat-model (`docs/actpass-bypass-threat-model-2026-06-13.md`):

> **The proxy is correct. Coverage is a credential-custody and network-egress problem, not a proxy-correctness problem.**

Every call that *reaches* the proxy is enforced properly. The upstream is resolved from server env and never client-supplied, so an agent cannot redirect the proxy. Every forward is SSRF-guarded. The credential is bound server-side from the encrypted vault and never returned to the caller or written to evidence. A non-`allow` / `warn` decision returns immediately with no execution — fail-closed. On `allow`, the replay guard runs *before* the credential is bound, and the decision is sealed into the hash-chained ledger after the upstream call.

So the engine does not leak. The enforcement claim collapses only if the agent can reach the upstream **without traversing the proxy**. That precondition is not enforced by any code in this repository — it is a property of *how the agent is deployed*. This is the single load-bearing assumption of the whole gateway model, and stating it plainly is a feature, not a weakness.

Bypass taxonomy. The ways an agent can route around a *voluntary* proxy are finite and enumerable (Appendix C carries the exhaustive treatment; the ranked summary):

#	Vector	Why it works
1	**Ambient credential in agent env** — PAT/token in an env var, `~/.config`, or OS keychain	Agent calls the upstream directly; proxy never invoked
2	**Shelling out to `git` / `gh`** with local creds over HTTPS/SSH	Bypasses the HTTP proxy entirely
3	**Second / ungated MCP server** carrying its own token	Parallel path with no ActPass in front — one ungated surface is a hole
4	**Agent writes and runs a script** using a readable on-disk token	Same as #1, one indirection removed
5	**SSH git remotes / agent forwarding**	Non-HTTP path to the upstream

The decisive mitigation is network egress control, not credential hygiene alone. If the agent host's egress policy permits *only* the ActPass proxy and blocks the upstream (e.g. `github.com`, `api.github.com`) directly, then even a leaked credential has nowhere to go — traffic has no other route. Egress control converts the proxy from a **voluntary** chokepoint into a **mandatory** one. This is a deployment pattern — a firewall rule, a Kubernetes `NetworkPolicy`, a VPC egress restriction — not a code change, and it is the headline of any pilot or self-hosted setup: strip ambient credentials, bind them only in the vault, lock egress, *then* measure coverage. Measuring coverage without those steps measures nothing.

7.5 Non-goals — what ActPass does not defend on its own

Candor about the boundary is part of the model:

- **Read actions are not gated** by default, deliberately, to keep read latency out of the enforcement path. Data exfiltration via read APIs requires a separate response-filtering layer.
- **Credential binding is per-server, not per-action.** A credential bound for a server scope covers any action within that scope until released; a confused-deputy *within scope* is possible. Binding granularity can be tightened where a partner requires it.
- **The secret is decrypted in gateway memory at execution time.** In managed hosting this occurs inside ActPass's own trust boundary; customers who require the secret to never leave their network use the Enterprise in-VPC self-hosted gateway (§4), which keeps decryption inside their VPC.
- **Egress enforcement itself lives outside ActPass code.** ActPass provides the mandatory chokepoint *if and only if* the deployment routes egress through it. That is the operator's responsibility, and we say so up front.
- **Policy authoring is a trust boundary.** The engine validates policy *structure*, not *intent*. A poisoned or careless policy draft is enforced faithfully — for example, a leading broad `allow` rule silently neuters the restrictive rules below it under first-match-wins evaluation, because the first matching rule decides and the tighter rules downstream never run. ActPass cannot tell an over-permissive policy from an intended one, so who may change a policy — and who reviews the change — is a trust boundary that lives with the operator. We recommend a human or second-control review gate on every policy change; an ordering-lint that flags a broad `allow` shadowing narrower rules is a possible future guard, not a current one.

The measured coverage number — the fraction of an agent's sensitive upstream actions that actually traversed the proxy — is empirical and can only be established in a real install, cross-checked against an independent ground truth such as the upstream provider's own audit log. §9 (Testing and Metrics) and §11 (Status and Roadmap) own the honest accounting of what has been validated in the field versus what the architecture is designed to guarantee.

8. Verifiability and Compliance

An evidence fabric that only its operator can read is not evidence — it is a log the operator can be asked to trust. ActPass is designed so that a skeptical third party (an auditor, a regulator, a design-partner's security team, or an MCP consumer who has never held an ActPass account) can validate the artifacts ActPass produces using nothing but published public keys and canonical bytes. No shared secret ever leaves the platform, and no round-trip through ActPass is required to check a signature. This section covers three surfaces: signed audit exports, the ActPass Certified badge, and how the mechanisms described elsewhere in this paper map onto external control frameworks.

8.1 Signed audit exports

Every attestation ActPass emits — trust reports, certification records, and the exported segments of the evidence ledger — carries an **Ed25519 detached JWS** signature. The signer lives in `lib/actpass/report-signing.ts` and is a thin, dependency-light wrapper over `jose` (the `EdDSA` algorithm, no custom crypto).

Two properties make these signatures independently verifiable:

- **Asymmetric, not shared-secret.** The signing key is a private Ed25519 JWK held only by ActPass (env `ACTPASS_REPORT_SIGNING_KEY_JWK`). Verification uses only the corresponding public JWK, which `getReportPublicJwk()` publishes with its `kid`, `alg`, and `use: 'sig'`. An HMAC scheme would force every verifier to hold the same secret ActPass uses to sign — meaning any verifier could forge. Ed25519 removes that: a partner verifies without ever being able to mint.
- **Detached payload, per RFC 7797.** Signatures are produced with `b64: false` and the `b64` value in the JWS `crit` header, so the payload bytes never travel inside the signature. The report artifact stays human-readable and is canonicalized exactly once — deterministic key-sorted JSON via `sortedStringify`, matching the manifest canonical form — and the verifier re-supplies those canonical bytes alongside the detached signature.

The verification API is deliberately total and pure. `verifyReportWith(payload, sig, publicKey)` returns a boolean and **never throws** on a bad signature; any tampering with the payload, the protected header, or the signature — or a signature from a different key — yields `false`. A third party runs exactly:

```
import { flattenedVerify, importJWK } from 'jose';
const key = await importJWK(published_public_jwk, 'EdDSA');
const ok = await verifyReportWith(record, signature, key); // boolean, offline
```

The signer also fails closed. In production, if `ACTPASS_REPORT_SIGNING_KEY_JWK` is unset, `getSigningKey()` refuses to sign rather than fall back to a hardcoded or ephemeral key — a default key would make every attestation forgeable across restarts. Outside production it mints a per-process key with a loud warning, and those signatures deliberately do not survive a restart.

The same Ed25519 primitive signs Action Passports (see §6), so passport minting, request-integrity signing, and attestation signing share one auditable keypair discipline rather than a scattering of ad-hoc secrets. One caveat about that discipline, though: it is not uniform across surfaces. Report, attestation, Certified-badge, and evidence-anchor signing all currently use a **single, non-rotating** Ed25519 key with a static `kid` — there is no `kid`-keyed overlap registry the way passport verification carries one. So rotating the report signing key without invalidating every previously issued attestation is a tracked gap, not a solved problem (see §11). The offline value proposition is unaffected: a third party still verifies any single attestation against the published public JWK with no round-trip and no shared secret. What is missing is the ability to retire that key and keep old signatures verifiable at the same time.

8.2 ActPass Certified

ActPass Certified turns the same signing and hashing primitives into a public trust signal for the MCP ecosystem. Any MCP or OpenAPI server author can run a **free, self-serve scan** of their tool manifest (`POST /v1/certified` , authenticated, tenant derived from the API key — never the request body). The manifest is scored by the existing `scanManifest` engine (see §6.5); if it clears the published thresholds, ActPass mints a signed `CertRecord` and seals its issuance into the evidence ledger. The feature is pure composition of primitives described elsewhere — no new crypto, no new scanner (`lib/actpass/certified.ts`).

Tiers and thresholds follow the severity-risk scale (lower score is safer), and any critical finding fails to certify — no badge is minted at all:

Tier	Threshold	Additional gate
Gold	`risk_score ≤ 20`	no `high` or `critical` finding
Silver	`risk_score ≤ 50`	no `critical` finding
Bronze	`risk_score ≤ 75`	no `critical` finding
— (not certifiable)	`risk_score > 75` , or any `critical` finding	badge withheld

The badge is served as a shields.io-style SVG (`GET /v1/certified/:id/badge.svg`) that authors embed in a README, alongside a public human report (`/certified/:id`) and a public verify endpoint (`GET /v1/certified/:id`) returning `{ record, signature, public_jwk }` . Three mechanisms — all reusing §8.1's primitives — make the badge hard to abuse:

- **The signature stops forgery.** The `CertRecord` (id, `spec: 'actpass-certified/1'` , `server_ref` , `manifest_hash` , `score` , `tier` , `highest_severity` , `issued_at` , `expires_at`) is the exact payload the detached JWS covers. Nobody can fabricate a Gold record for a server ActPass never scored, because they cannot produce a valid signature.
- **The hash binding stops badge-borrowing.** The record is bound to `manifest_hash = hashManifest(canonicalizeManifest(manifest))` , the same tamper anchor used for drift detection. A different or modified server hashes differently, so a borrowed badge cannot claim to certify code the record never covered.
- **Every mint is sealed as evidence.** Each issuance writes a `certification_issued` event via `sealAndInsertEvidence` , so double-mints and re-issues are visible in the tenant's hash-chained ledger.

Certification is not static. `verifyCert(record, signature, currentManifestHash?)` resolves display status against the shared `BadgeStatus` palette: a valid, fresh cert shows its tier; an expired cert (certs carry a fixed TTL, forcing periodic re-scan) flips to **Expired**; and when a re-scan or live probe supplies a `currentManifestHash` that differs from the bound hash, the badge flips to **Drift** — the "rug-pull" case where a server that earned a badge later changed its tools. Because the badge is embedded in public READMEs, the badge routes fail *open* to a neutral `Setup` badge on any lookup error (a 5xx would render broken images across every consumer), while the underlying verification always fails *closed*.

Architecturally this is a viral trust loop for the MCP ecosystem: an author embeds a badge to signal safety, every consumer who renders it is one public `GET` away from verifying it against ActPass's published JWK, and drift or expiry degrades the signal automatically without any consumer action. The scan certifies the *declared* manifest, not live wire behavior; pairing certification with a continuous live-probe feed is the named upgrade path, and `verifyCert` already accepts the live hash it needs.

8.3 Standards mapping

ActPass's mechanisms map cleanly onto the control frameworks a security architect evaluates against. The table below is a mapping of *architecture to obligation*, not a claim of certification. **ActPass holds no third-party audit today** — SOC 2, ISO 27001, and formal AI Act conformity assessment are roadmap items (see §11), and the

deterministic-decision-plus-signed-evidence design exists in part to make those audits inexpensive to pass, not to assert their result.

Framework	Obligation	ActPass mechanism
NIST AI RMF	GOVERN — documented, enforceable policy	Deterministic JSON-DSL policy engine, fail-closed, versioned per tenant (§6.3)
	MAP / MEASURE — risk characterization	Manifest scanner, 21 action categories, risk tiers, 60 typed reason codes (§5, §6.5)
	MANAGE — control + traceability	Preflight allow/deny/require-approval outside the LLM; hash-chained evidence ledger (§5, §6.7)
EU AI Act (high-risk systems)	Art. 12 — automatic record-keeping / logging	Append-only, hash-chained evidence fabric with signed exports (§6.7, §8.1)
	Art. 14 — human oversight	Human-in-the-loop approval FSM with SLAs and RBAC-scoped deciders (§6.6)
	Art. 15 — accuracy, robustness	Fail-closed determinism; passport TTL clamp; SSRF and rate-limit guards (§6.1, §7)
OWASP LLM Top 10	LLM01 Prompt Injection / LLM06 Excessive Agency	Authorization rendered *outside* the model; a poisoned prompt cannot mint a passport or widen a policy (§2, §5)
	LLM08 Vector/Manifest tampering	Manifest-hash binding + drift detection; Certified badge flips on change (§6.5, §8.2)
MCP spec / zero-trust	Verifiable tool identity, least privilege	Per-tool manifest identity, scoped short-lived passports, deny-by-default; API-key-is-identity (§4, §6.2)

Each row is grounded in a mechanism this paper documents in detail. The honest maturity of each — built, live-validated, or planned — is the subject of §11.

9. Validation and Engineering Rigor

A deterministic authorization decision is only as trustworthy as the evidence that it behaves as specified under adversarial input. ActPass earns that trust through an architectural choice that makes the enforcement core testable in isolation, and a testing methodology that encodes each attack from the adversary model (§7) as an explicit, runnable invariant. This section describes that methodology and reports the measured state of the suite. It scopes those results honestly: the validation described here is **code-level and hermetic**. End-to-end validation against live external systems — a real database walkthrough, IdP round-trips, live SIEM ingestion — is a separate and still-open workstream, accounted for in §11.

9.1 A pure enforcement core over an injected repository

The preflight engine (§5) is written as a pure function over an injected `PreflightRepository` interface. All durable state the decision needs — passport records, replay claims, revocations, tool manifests, approval verification — is reached through that interface rather than through a direct database call. Two implementations satisfy it:

- `InMemoryPreflightRepository` backs the entire test suite. It holds state in plain data structures, so a test can construct any world — a revoked passport, a drifted manifest, a cross-tenant record — as an in-process fixture, with no Postgres, no network, and no fixtures to seed or tear down.
- A Drizzle-backed repository (`lib/actpass/db-repo.ts`) backs the live route in production, reading from the 34-table Neon Postgres schema.

Because the decision logic is identical across both — the same `preflight()` code path runs in a test and in production, differing only in which repository object it was handed — an invariant proven against the in-memory repository is a property of the code that ships. Runtime stores (replay cache, revocation store, approval verifier) are

injected the same way and default to fail-closed in-memory behavior. The consequence is a core that runs the full policy / passport / drift / evidence pipeline **without a database**, which keeps the suite fast and, more importantly, hermetic: a test's outcome depends only on the world the test constructed.

9.2 Red-team-as-tests

This hermeticity is what makes the central methodology possible: **every attack vector is an assertion.** Rather than describing bypass resistance in prose, each entry in the bypass taxonomy (§7) — passport replay, cross-tenant transplant, forged approval hashes, scope escapes, manifest rug-pulls, missing-claim fail-open — is encoded as a test that constructs the malicious input and asserts the specific fail-closed outcome: the exact `Decision`, the exact `ReasonCode`, the exact HTTP status. A dedicated red-team suite concentrates these adversarial cases; correctness-focused suites cover the honest paths. When the engine was subjected to a multi-agent adversarial review, every confirmed finding — a fail-open `!=` on a missing claim, a revocation time-of-check/time-of-use gap, manifest-drift holes, incorrect 403 statuses, argument-redaction leaks, a fail-open network fetch — was fixed **and pinned by a regression test**, so the same class of defect cannot silently return.

The worked example is the flagship refund flow, exercised across its full decision surface: a small refund allows, a medium refund routes to `require_approval`, a large refund denies, a drifted tool manifest forces `require_tool_reapproval`, a wrong-tenant passport returns 403, a missing passport on a critical tool denies, a replayed passport denies, and an evidence-write failure fails the whole decision closed rather than allowing an unlogged action.

9.3 Measured state of the suite

The numbers below are measured from source at the current head.

Metric	Value
Enforcement engine	**13,685 LOC** across **67** `.ts` **files** (`lib/actpass/`)
Vitest test files	**143** (112 unit + 31 integration)
Vitest test cases	**1,127** across **305** `describe` blocks
Playwright E2E	**9 spec files**, **45** test cases (DB-backed paths gated on `E2E_REAL_DB=1`)
Typed reason codes asserted against	**60** (`ReasonCode` union)
Data model under test	**34** tables, **25** migrations
API surface	**64** routes (**43** under `/v1`), **22** OpenAPI-documented paths
Test runner	Vitest v3, Node env; typecheck (`tsc --noEmit`) CI-enforced

The unit layer covers the pure modules — passport issue/verify/replay, the policy DSL and its packs, manifest canonicalize/hash/diff/drift, the evidence hash chain, RBAC, the AES-256-GCM credential envelope, OIDC verification, reason-code stability, and the red-team cases. The integration layer covers the orchestrated `preflight()` pipeline end to end over the in-memory repository. The 60 typed reason codes are not decoration: because each is a stable identifier, tests assert on them directly, which turns "the engine denied for the right reason" into a checkable fact rather than a judgment call.

9.4 Policy expressiveness, proven not asserted

A recurring skeptical question is whether a deterministic JSON DSL — no LLM in the decision path — is expressive enough for real customer policies, or whether ActPass will be forced onto a general policy language like Rego. This was answered empirically. Three realistic customer policies were encoded in the **actual** engine (`lib/actpass/policy.ts`) and run against realistic contexts; the result is a passing self-test (`tests/unit/actpass-policy-expressiveness.test.ts`). The operator set — `== != > >= < <= in not_in contains matches`, plus `$ref` cross-path comparison against passport constraints, `all` / `any` groups, first-match-wins ordering, and a fail-closed default deny — expressed every **decision** in all three policies.

The self-test also names the DSL's real boundaries honestly, and shows they are not language limitations. Two require an **enricher** — a per-tool gateway adapter that populates the `PolicyContext` before evaluation — to supply derived inputs (CI status, protected-path globs, business-hours flags) or stateful counters (per-agent daily deploy counts aggregated from tables already in the schema). The engine deliberately does no I/O, so these are inputs, not new grammar. A third apparent gap — response-side row/field limits — is not an engine gap at all but a different enforcement point, since a request-time gate cannot inspect a response that does not yet exist. Critically, every gap **fails closed**: the `!= true` idiom means an absent signal takes the restrictive branch, never a silent allow. The conclusion the self-test supports is bounded and testable: the DSL is sufficient for pilots, with the remaining risk being field-confirmation that specific design-partner policies fit this shape — encode their top policies on day one and find out.

9.5 End-to-end walkthrough of the deployed product

Hermetic tests prove the core; they do not prove that the deployed product a partner actually sees is coherent. That gap is closed by an automated production walkthrough: an authenticated browser session traverses all 34 dashboard and public routes of the live deployment, recording for each the HTTP status, the rendered page, a full-page screenshot, and the browser console. The most recent run — executed after the ServiceNow integration merge — returned HTTP 200 on every route with zero console errors, and its artifacts (screenshots, per-route headings, `report.json`) are versioned in the repository under `walkthrough/`. This is deliberately scoped: it validates the deployed surface, not the live third-party round-trips that §11 declines to claim. What it does make reproducible is a claim most decks cannot support — that the product in the screenshots is byte-for-byte the product that is deployed.

10. Deployment and Integration Surface

ActPass is delivered as a managed service. There is no gateway to install, no cluster to operate, and no self-hosted binary to secure; every client is a thin edge that talks to one hosted control plane. This section describes the adoption surface a design partner actually touches: the base URL, the identity model that collapses configuration to a single secret, the client libraries and integrations, and the distribution contract each artifact enforces on itself. The deployment topology and plane separation are owned by §4; the honest live-versus-pending accounting is owned by §11 and only summarized here.

10.1 One base URL, one secret

The managed API has a single canonical base URL, resolved from one constants module and overridable by a single environment variable:

```
ACTPASS_API_BASE_URL = https://www.api.actpass.org   # passport issuer + gateway
ACTPASS_WEB_BASE_URL = https://www.actpass.org        # dashboard, approvals, verify pages
```

Every client — TypeScript SDK, Python SDK, CLI, integrations — imports these defaults rather than hardcoding a host, and a guard test (`tests/unit/actpass-canonical-hosts.test.ts`) asserts the product owns `actpass.org` and never `actpass.com`.

Configuration reduces to pasting one API key. This is a direct consequence of the **API-key-is-identity** contract: the server resolves both the tenant and the agent from the authenticated bearer key (`apdv_…` developer key or `apk_…` agent/gateway key) and explicitly ignores any client-supplied `x-actpass-tenant`. A request carrying an agent key is scoped to that agent's registered tools and policies; the tenant is derived from the principal, never from the request body. Consequently `tenantId` and `agentId` in client config are optional hints kept for telemetry and back-compat — they are never required and never trusted for authorization. The minimal integration is therefore: mint a key in **Settings → API Keys**, paste it, done.

10.2 The client surface

Channel	Package / ref	What it does
TypeScript SDK	`@actpass/sdk`	`enforcePolicy()` / low-level `createActPass().guard()`
Python SDK	`actpass` (PyPI)	Same preflight/enforce shape for Python agents
CLI	`@actpass/cli`	Local `scan`, enforcement proxy, browser-attested enrollment
MCP proxy	CLI / gateway mode	Sits in front of an MCP server; preflights every tool call
REST/OpenAPI proxy	gateway mode	Same, for REST tool surfaces described by OpenAPI
GitHub Action	`actpass/scan-action`	Scan + manifest-drift gate in CI
n8n node	`n8n-nodes-actpass`	Preflight inside n8n workflows
Managed integrations	Slack · Teams · ServiceNow · Vanta/Drata	Approvals routing; compliance export

The SDK is the primary path. Its core call wraps a risky tool invocation and returns a deterministic decision from the managed cloud, failing closed on any non-2xx, auth failure, or network error:

```
import { createActPass } from '@actpass/sdk';

// The API key IS the identity — server resolves tenant + agent from it.
const client = createActPass({ apiKey: process.env.ACTPASS_DEVELOPER_KEY! });

const { decision, result } = await client.guard({
  goal: 'resolve_refund_request',
  tool: 'stripe.refund.create',
  args: { chargeId: 'ch_123', amount: 14900 },
  execute: async () => stripe.refunds.create({ charge: 'ch_123', amount: 14900 }),
});
// execute() runs ONLY when decision.decision is 'allow' or 'warn'.
```

The `guard()` wrapper is the enforcement invariant made ergonomic: the supplied `execute` callback is dispatched only on an `allow` / `warn` decision, so a blocked or approval-pending action cannot reach the tool by accident. The Python SDK mirrors this shape. The lower-level path is the raw preflight, identical to what every other client sends:

```
curl -X POST https://www.api.actpass.org/api/v1/actions/preflight \
  -H "Authorization: Bearer apk_YOUR_KEY_HERE" \
  -H "Content-Type: application/json" \
  -d '{ "tool": "stripe.refund.create",
        "resource": "stripe:charge:ch_123",
        "args": { "amount": 4900, "currency": "usd" },
        "agent_id": "support_agent", "user_id": "user_456",
        "mode": "enforce" }'
```

On a fresh workspace the response is `{"decision":"deny","reason_code":"policy.denied_default", …}`. That deny is the intended signal: no policy has matched yet and the gateway fails closed. Every response carries a typed `reason_code` (see §5) and a `chain_id` linking it into the evidence ledger (§6).

The **CLI** offers a no-account local `scan` of MCP/OpenAPI/n8n manifests, a local enforcement proxy, and one-step enrollment via browser-attested pairing (`actpass install`) or an already-minted key (`actpass login --key apk_…`). Credentials land at `~/.actpass/credentials.json` (mode `0600`), and all commands default to the same base URL. Deployed inline, the CLI and gateway run as an **MCP proxy** or **REST/OpenAPI proxy**: they intercept each tool call, run the preflight pipeline, and forward only allowed actions — so an unmodified agent gains enforcement with no code change.

For the **CI and workflow surface**, the GitHub Action runs the same static scan and fails the build on manifest drift (a tool whose meaning changed since last consent — see §6), giving a regression gate on tool-manifest risk. The **n8n community node** brings preflight into low-code workflows. The **managed integrations** — Slack and Teams for human approvals (§6), ServiceNow for change routing, and Vanta/Drata for compliance export (§8) — run OAuth and state inside the managed cloud, so a partner adds them without standing up any customer-run server.

10.3 Every artifact carries a guard

Each published artifact ships its own `README.md` as its marketplace landing page and, per the project's distribution contract, leaves exactly one runnable check that fails if that contract breaks — for example a pack-and-`require` test proving the tarball imports, or an assertion that the client's default URL equals the single host constant. The distribution contract is thus self-testing: a divergent hostname or a broken package export trips a test rather than reaching a partner as a silent fail-closed `BLOCKED` .

Channel maturity is uneven and stated honestly in §11: the SDKs and CLI are the load-bearing, guard-tested paths; the GitHub Action and n8n node are built and pending one-time marketplace/registry registration; and the Slack, Teams, ServiceNow, and Vanta/Drata integrations are managed-cloud channels whose live third-party round-trips are not yet claimed as validated. Managed-hosting-only is deliberate: self-hosting and Docker distribution of the gateway are explicit non-goals, which is what keeps the adoption surface to one URL and one key.

10.4 The six-line integration

For an agent that is code, the entire adoption story is one wrapped call:

```
import { ActPass } from '@actpass/sdk';

const actpass = new ActPass({ apiKey: process.env.ACTPASS_DEVELOPER_KEY! });

const decision = await actpass.enforcePolicy({
  toolName: 'stripe.refund.create',
  args: { chargeId: 'ch_123', amount: 14900, currency: 'USD' },
});

if (decision.status !== 'ALLOWED') throw new Error(decision.reason);
// ...only now does the refund execute...
```

One key, no tenant or agent wiring — the key **is** the identity (§3.4) — and any transport or auth failure returns `BLOCKED` , never a silent allow (§3.2). For stronger structure, the lower-level `guard()` accepts the tool invocation as a closure and runs it only on an `allow` / `warn` decision:

```
const { decision, result } = await client.guard({
  goal: 'resolve_refund_request',
  tool: 'stripe.refund.create',
  args: { chargeId: 'ch_123', amount: 14900 },
  execute: async () => stripe.refunds.create({ charge: 'ch_123', amount: 14900 }),
});
```

Because execution lives inside the guard, there is no code path in which the agent fires the tool without a decision — the difference between an enforcement boundary and a suggestion.

This shape yields a canonical three-beat demonstration that exercises the whole system in under two minutes: run a small refund and watch it pass frictionlessly (`allow`); rerun the same code at $50,000 and watch the agent halt (`require-approval`); then open the dashboard, where the identical action is pending in the Approval Center and already sealed into the hash-chained evidence ledger. Allow, block, evidence — with no logging code written by the integrator.

Two items have since moved from *built* to *exercised*. The ServiceNow integration is merged to `main` and surfaced in the product's integration catalog, and the full production walkthrough of §9.5 has been re-run against the deployed product — 34 routes, HTTP 200 on each, zero browser-console errors — with its screenshots feeding the design-partner demo collateral directly. Neither changes the honest cap above: the walkthrough validates the deployed surface, not a live third-party round-trip, and commercial confidence still waits on a signed pilot.

11. Status and Roadmap

This section is the honest accounting. ActPass is at **late-alpha maturity**: the enforcement core is production-grade and independently re-verified, the surrounding product surface is built but not yet exercised against live external systems, and a small, well-bounded set of gaps remains before general availability. Rather than a single completion percentage, we separate three states — **live-validated**, **built but not externally validated**, and **planned / known gaps** — because conflating them is exactly the misrepresentation a skeptical buyer is right to distrust. Every claim below is grounded in internal audit and readiness reviews (`AUDIT_REPORT.md` , `MEMO_RECONCILIATION.md` , `docs/actpass-production-readiness-2026-07-01.md` , `docs/actpass-go-no-go-decision-2026-06-13.md` , and the launch-readiness assessment).

We do **not** claim a signed enterprise customer in production, a completed SOC 2 audit, a live SIEM round-trip against a customer's SIEM, or a live IdP/SSO round-trip. Those are roadmap items. The most recent adversarial readiness review verified every load-bearing claim at file:line and performed **no live-systems run** — confidence in the static-source findings is high, but live confirmation remains the correct first post-fix check.

11.1 Live-validated: the deterministic core

These primitives are pure, deterministic, fail-closed, and covered by the hermetic red-team test suite (§9). Independent adversarial re-checks re-opened each claim in source and confirmed them; the credential-vault dimension scored **8.5/10** and access control on hot paths **8.0/10** in the most recent review — the strongest dimensions in the product.

- **Deterministic preflight engine** (`lib/actpass/preflight.ts`) — pure, injectable, server-trust-only, defaulting to `deny` and failing closed on storage, policy, manifest, or evidence errors. No LLM sits in the decision path.
- **Action Passport crypto** (`lib/actpass/passport.ts`) — EdDSA/Ed25519 via `jose` with `kid` , full issuer/audience/tenant/agent/user/ `exp` / `jti` /drift verification, and single-use replay with a TOCTOU re-check. Verification is **fully offline** — an in-memory public-key map and local `jwtVerify` , no network call — the property that makes a customer-VPC gateway the enforcement root.
- **Deterministic policy evaluator** (`lib/actpass/policy.ts`) — first-match-wins, fail-closed on every branch, proven expressive enough for three realistic customer policies with no external policy engine (OPA) required.
- **Manifest-drift algorithm** (`lib/actpass/manifests.ts` , `drift.ts`) — canonicalization plus per-schema and global hashing feeding a faithful drift classifier; production-grade as a pure function.
- **Evidence hash chain** (`lib/actpass/evidence.ts`) — canonical-JSON SHA-256 with per-chain linkage and completeness scoring to the specified weights. **The append path is atomic** — DB unique constraints plus bounded retry close the earlier concurrent-writer fork (audit finding H-22).
- **No-client-trust discipline** — `tenantId` is always derived from the authenticated principal (`lib/actpass/api-auth.ts`), never the request body. All 22 documented query routes are correctly team-scoped; a cross-tenant

ciphertext-transplant attempt against the AES-256-GCM vault (AAD-bound to tenant+provider) was reproduced and **blocked**.

- **Inbound integration authentication** — Slack constant-time HMAC over the raw body with a replay window, Teams Bot Framework JWT verification with issuer/audience pinning, and gateway-scoped keys for ServiceNow. These auth *checks* are verified in source; what is not yet validated is the *live round-trip* against the real external systems (§11.2).

Security-remediation trajectory. The bulk of the earlier audit's 9 CRITICAL and 30 HIGH findings are closed and covered by named regression tests: free-tier Enterprise escalation (C-1), checkout-session binding (C-7), the missing `subscription.created` webhook (C-8), the approval TOCTOU (C-9 via optimistic-lock CAS), the cross-chain replay scoping (C-6), unauthenticated passport minting (C-2), the LLM-scope allowlist (C-3), SSRF on webhook URLs (H-5), the coloring blind spot (H-19), and the signed-export fail-open (now returns HTTP 500 fail-closed). The two hard-stops surfaced in the July re-audit are now closed as well: the **string-typed `amount` money-ceiling bypass** is fixed by `toFiniteNumber` coercion that fails closed on a non-numeric present amount at the passport, policy, and credential layers (`lib/actpass/passport.ts`, `policy.ts`, `credentials.ts`), and the **evidence `event_id` recompute mismatch** is fixed by excluding `event_id` from `computeEventHash` (`lib/actpass/evidence.ts`) so a genuinely sealed chain re-verifies as valid. Both are pinned by regression tests, including the real-DB seal→reload→recompute test `tests/integration/actpass-evidence-seal-reload.test.ts` (gated on `ACTPASS_TEST_DB=1`). The trajectory is real and green in the suite.

11.2 Built, not yet validated against live external systems

These are implemented and unit-tested but have not completed a round-trip against the real third party, a customer VPC, or a package registry. They are honest "built" — not "proven in the field."

Capability	State	What validation remains
Dashboard end-to-end	Backend real; funnel breaks at the API-key step (settings page served a fake key)	Wire the page to the existing `POST /v1/keys`; run one full onboarding pass
Real IdP / SSO round-trip	Password auth works; no SSO/OIDC on the primary flow	Live SAML/OIDC integration with a real IdP
Live Slack / Teams approval loop	Inbound decide path and Block Kit builder are real	An **outbound** notification on approval creation; no reviewer is notified today
SIEM export	JSON/CSV/MD/JSONL exporters and Ed25519 signing exist	Ingestion against a real SIEM
Self-hosted / customer-VPC gateway	`ACTPASS_ROLE=gateway` image, Helm workload, offline JWKS, local credential binding exist	A design-partner install; interception-coverage measurement
Distribution channels	SDK/CLI/PyPI/GitHub Action/n8n node built and publish-ready	One-time registration/publish (npm/PyPI/Marketplace, OAuth app registrations) and version alignment

The container-image and distribution mechanics carry known point-in-time defects: the primary `Dockerfile` references `pnpm` while the repo ships `package-lock.json` (a trivial `npm ci` fix), and a point-in-time registry check found the published CLI/SDK behind the repo version. These are wiring and release-ops items, not architecture.

11.3 Planned / known gaps being closed

- **Compliance sampling across all chains.** The CC-2 control samples a single most-recent chain; sampling is disclosed (`sample_chain_id`), so it under-covers rather than misreports — but it is not audit-grade until it aggregates across all chains. With the §11.1 evidence-recompute fix landed, this is the live coverage gap.
- **PII minimization for Vanta/Drata export.** An `evidenceFilter` allowlist to strip PII before third-party export.
- **SCIM** provisioning (on the SSO/OIDC roadmap).

- **Tightened approval-reuse semantics** and payload-binding hardening.
- **CSP** `script-src` tightening and migration of serial-integer primary keys to UUID/ULID for enumeration resistance.
- **Persistent signing-key fail-closed** — with `ACTPASS_SIGNING_KEY_JWK` unset in production the passport bootstrap currently warns and mints an ephemeral key rather than refusing to boot; boot-time validation must fail closed, mirroring the credential key which already does.
- **Signing-key denylist propagation.** The per-jti revocation store is DB-backed (`DbRevocationStore`), but the verification-key registry and its denylist are process-local in-memory (`passport.ts` : `revokeVerificationKey` mutates in-process maps only). Denylisting a compromised signing `kid` therefore does not yet propagate across serverless replicas or survive a cold start; a parallel durable store for the key denylist is needed to make revocation authoritative fleet-wide.
- **Report/badge/anchor signing-key rotation.** Attestation, report, Certified-badge, and evidence-anchor signing use a single non-rotating Ed25519 key (`report-signing.ts`, one static `kid`), with no kid-keyed overlap registry of the kind passports already have. Rotation with an overlap window is tracked but not yet implemented, so a key roll today would break verification of previously signed artifacts.
- **Credential DEK custody.** The credential vault's master data-encryption key is a single env-supplied DEK (`ACTPASS_CREDENTIAL_KEY`, with `ACTPASS_CREDENTIAL_KEY_OLD` for retired-key overlap during rotation). Per-tenant, KMS/HSM-backed keys are roadmap. DEK custody is the trust root: a leaked DEK decrypts every tenant's secrets until KMS-backed keys land, so this is the highest-consequence key-management gap.

11.4 Roadmap to GA and to a self-hosted pilot

The path to GA is **wiring against existing interfaces, not re-architecture** — the readiness review found most remaining fixes rated Low or Trivial. Sequenced security-first:

1. **Remaining GA gate:** add signing-key fail-closed validation and enforce the `runtime_enforcement` entitlement on the action routes. The two correctness hard-stops are already closed and regression-covered (the `amount` finite-number coercion and the evidence `event_id` recompute fix, §11.1); the remaining action is running the DB-gated integration suite — including `tests/integration/actpass-evidence-seal-reload.test.ts` — in CI, not writing it.
2. **Prove it runs:** fix the Dockerfile to npm and add a docker-build CI step; enable the real-DB integration tests in CI (they exist behind `ACTPASS_TEST_DB=1`); run one genuine live smoke pass — real DB, real keys, one preflight→execute→approval→export.
3. **Adoption plumbing:** wire the settings page to the real key-minting endpoint (reveal-once); add one outbound approval-notification hop; aggregate compliance verification across chains.
4. **Distribution:** publish the aligned CLI/SDK and either ship or remove the channels so docs match reality.

The **self-hosted pilot** dependencies are the four stateful checks behind injectable interfaces. Passport verification and replay are already gateway-local; approval is carried in the signed passport and fails closed; the one genuinely missing piece is **revocation-sync** — a pushed/polled denylist that fails closed on staleness. The intended model is a design parameter, not an open question: revocations propagate on a bounded push/poll interval, and the gateway fails closed once its last-sync age exceeds a configured staleness threshold, so the revocation-latency window is bounded by that threshold rather than unbounded. The local single-use replay store must likewise be **durable and shared across gateway replicas and restarts** — an in-process store would let a replayed passport slip through a second replica or a restart — so it is backed by shared state at the pilot rather than kept in memory. The exact interval and staleness threshold are to be finalized with the first pilot, but they are stated design bounds. Key distribution, revocation sync, the durable replay store, and evidence buffering are all implementations against interfaces that already exist. The honest cap on commercial confidence is not code quality but the absence of a signed design-partner pilot; the engineering foundation is real, tested, and ready to carry a first customer install.

12. Conclusion

The argument of this paper reduces to a single claim: enforcement belongs outside the model. An agent that reads context, decides, and calls a tool holds the standing authority of the human it acts for, exercised at machine speed (§2). A probabilistic guardrail cannot bound that authority, because a control you cannot reproduce you cannot audit, test, or defend. ActPass moves the authorization decision out of the language model and into a deterministic evaluator: given the same action — tool, resource, arguments, agent, tenant, mode, policy version — the same `allow` / `deny` / `require-approval` decision, every time, tagged with one of 60 typed reason codes (§3.1, §5). Determinism is what turns agent governance from something you hope holds into something you can prove.

Four guarantees carry that thesis, and each is grounded in source rather than asserted. The preflight engine is a **pure, fail-closed function** — any condition it cannot positively resolve becomes a deny, at every layer, by architectural mandate rather than a configurable toggle (§3.2, §6.1). Identity is the **authenticated key, never a client field**: tenant and agent are resolved from the bearer principal and re-derived server-side, which collapses cross-tenant and confused-deputy attacks into non-issues (§3.3, §3.4, §7). Attestations are **verifiable without shared secrets**: Action Passports and Certified badges are Ed25519 signatures a third party checks against a published public key, offline, with no ActPass in the loop — the property that lets a customer-VPC gateway act as its own enforcement root (§3.5, §6.1, §8). And the **evidence ledger is a hash-chained, canonical-JSON SHA-256 append-only record** whose atomic append path closes the concurrent-writer fork, giving a signed export a skeptic can replay independently (§6.5, §8, §9).

Section 11 owns the honest accounting, and this conclusion inherits its discipline: ActPass is at late-alpha maturity. The deterministic core is production-grade and independently re-verified; the surrounding product surface is built but not yet exercised against live external systems. We do **not** claim a signed enterprise customer in production, a completed SOC 2 audit, a live SIEM round-trip, or a live IdP round-trip — those are roadmap items. Two correctness hard-stops that once threatened those claims — a string-typed `amount` that bypassed the money ceiling, and an evidence recompute that mismatched on `event_id` — are now closed and regression-covered: the amount coercion fails closed on a non-numeric present amount across the passport, policy, and credential layers, and `event_id` is excluded from the evidence hash, guarded by a real-database seal-reload-recompute test. What remains pre-GA is broader validation against live external systems and parts of the product plane, not the deterministic core's correctness. The path to GA is wiring against interfaces that already exist, not re-architecture (§11.4). The one load-bearing assumption of the whole model — that egress is routed through the proxy — is a deployment property, and we state it plainly rather than hide it (§7.4).

This is where design partners come in. The engineering foundation is real, tested, and ready to carry a first install; what it lacks is a real network, a real SIEM, a real IdP, and a real adversary. A pilot supplies exactly those, and in return earns a governance layer whose every decision is deterministic, whose every action is evidence, and whose every attestation survives contact with a skeptic. We invite you to route one agent through it and measure what traversed the gate.

Appendix A — Preflight API and Reason-Code Reference

This appendix documents the wire contract of the core decision endpoint. It is grounded in `public/openapi.yaml` (spec `version: 1.1.1`) and the decision engine in `lib/actpass/preflight.ts`. See §5 for the decision pipeline itself; this section is the reference shape and reason-code index a caller integrates against.

A.1 Endpoint

```
POST /v1/actions/preflight
Authorization: Bearer <gateway-api-key>
Content-Type: application/json
```

Authentication is the gateway API key (created via `POST /v1/keys`; a development fallback reads `ACTPASS_API_KEYS`). All endpoints are tenant-scoped by the authenticated key — the caller never supplies `tenant_id` on this route. Under API-key-is-identity, the key resolves the tenant server-side, and the engine ignores any client-asserted tenant/tool-hash/approval/risk fields (`preflight.ts`, §11.2 hard rule).

A.2 Request body (`PreflightRequest`)

Field	Type	Required	Notes
`tool`	string	yes	Tool identifier, e.g. `stripe.refund.create`
`resource`	string	yes	Target resource URN, e.g. `stripe:charge:ch_123`
`agent_id`	string	yes	Calling agent identity
`user_id`	string	yes	End user on whose behalf the action runs
`args`	object	no	Action arguments; canonicalized and hashed server-side
`agent_name`	string	no	Used to resolve the applicable policy
`goal`	string	no	Declared task intent
`mode`	enum	no	`monitor` / `warn` / `enforce` / `strict`; may only *raise* enforcement above the tenant-authored policy mode, never lower it
`passport`	string	no	Compact JWS Action Passport (see Appendix B)
`audience`	string	no	Expected passport audience; defaults to the tool manifest server origin
`idempotency_key`	string	no	Becomes the evidence `chain_id` linking preflight and later execute

A.3 Response body (`PreflightResponse`)

The public schema exposes the decision and its audit linkage; the internal engine additionally returns `risk_tier`, `tool_manifest_hash`, `policy_hash`, and a structured `explain` block (`preflight.ts`).

Field	Type	Notes
`decision`	enum	`allow` / `deny` / `warn` / `require_approval` / `require_tool_reapproval`
`reason_code`	string	Typed reason (see A.4)
`evidence_event_id`	string	Sealed event appended to the ledger for this decision
`chain_id`	string	Evidence-chain id; `execute()` appends to the same chain
`approval_request_id`	string	Present only when `decision = require_approval`; the id to resolve in the approval queue
`risk_tier`	enum	`low` / `medium` / `high` / `critical`
`tool_manifest_hash`	string	Approved manifest hash the decision was evaluated against
`policy_hash`	string	Hash of the governing policy version
`explain`	object	`{ summary, matched_rules[], next_steps[] }`

Decisions are rendered under HTTP 200 in the normal case. Passport authentication failures map to 401 (missing/unverifiable) or 403 (mismatch, out-of-scope, replay); storage or evidence-write failures fail closed with an HTTP 500 body carrying a `policy.missing` or `evidence.write_failed` reason.

A.4 Reason codes (representative)

The `ReasonCode` union defines **60** typed codes across eight namespaces (`lib/actpass/core.ts`). The table below is a representative — not exhaustive — index grouped by decision stage; every typed code shown is emitted directly

by `preflight.ts` or is part of the `passport.*` / `tool.*` families it consumes. In addition to the typed 60, the engine emits a small number of raw runtime reason strings at preflight time that live outside the `ReasonCode` union — notably `circuit_breaker.tripped` (failure-rate breaker trips) and `policy.untrusted_code_execution` (action authority sourced from untrusted content).

Category	Representative codes	Emitted when
Passport / auth	`passport.missing`, `passport.invalid_signature`, `passport.expired`, `passport.not_yet_valid`, `passport.revoked`, `passport.audience_mismatch`, `passport.tenant_mismatch`, `passport.agent_mismatch`, `passport.user_mismatch`, `passport.tool_not_allowed`, `passport.resource_out_of_scope`, `passport.replay_detected`	Passport verification / scope / replay checks fail (deny; 401 or 403)
Tool / manifest drift	`tool.unknown`, `tool.manifest_changed`	Tool not in the approved registry (deny); manifest drifted and awaits reapproval (`require_tool_reapproval`)
Tool identity	`tool.*` (descriptor pin), `circuit_breaker.tripped`	Use-time descriptor or publisher signature fails; failure-rate breaker trips (deny; 403)
Policy — deny	`policy.denied_default`, `policy.missing`	Default deny with no matching allow rule; no policy governs the tool (fail-closed)
Policy — provenance	`policy.action_open_delegation`, `policy.untrusted_code_execution`	Action authority sourced from untrusted content (`require_approval`, or `warn` in monitor)
Approval states	`approval.satisfied`	Policy required approval and a verified `approval_hash` in the passport satisfies it (upgraded to `allow`)
Evidence / system	`evidence.write_failed`	Pre-write to the ledger fails in `enforce` / `strict` (deny; 500)

The `approval.*` family (5 codes total) and `args.*` (4), `refund.*` (3), and `credentials.*` (2) families cover downstream approval-queue transitions and pack-specific rules; consult `lib/actpass/core.ts` for the full enumeration. The full breakdown is `tool.*` 23, `passport.*` 13, `policy.*` 9, `approval.*` 5, `args.*` 4, `refund.*` 3, `credentials.*` 2, `evidence.*` 1.

A.5 Worked example

Request — an AI support agent proposing a refund with a valid passport:

```
POST /v1/actions/preflight
Authorization: Bearer ak_live_...
{
  "tool": "stripe.refund.create",
  "resource": "stripe:charge:ch_123",
  "args": { "amount": 4200, "currency": "usd", "customer_id": "cus_42" },
  "agent_id": "agent_support_01",
  "agent_name": "support-copilot",
  "user_id": "u_987",
  "goal": "Refund duplicate charge for ticket #5521",
  "mode": "enforce",
  "passport": "eyJhbGciOiJFZERTQSIsImtpZCI6...",
  "idempotency_key": "refund-5521-a1"
}
```

Response — policy required approval, and a verified `approval_hash` carried in the passport satisfies it:

```
{
  "decision": "allow",
  "reason_code": "approval.satisfied",
  "risk_tier": "high",
  "tool_manifest_hash": "sha256:9f2c...e11",
  "policy_hash": "sha256:41ab...77d",
  "evidence_event_id": "evt_01J8...",
  "chain_id": "refund-5521-a1",
  "explain": {
    "summary": "Policy refund_policy v3: approval requirement satisfied by verified approval_hash.",
    "matched_rules": ["allow_small_refund"],
    "next_steps": ["proceed_to_execute"]
  }
}
```

Had the passport carried no approval grant, the same request would return `decision: "require_approval"` with `reason_code: "policy.approval_required"` and a populated `approval_request_id`, routing a reviewer to the queue (§6.4) before the action may proceed.

Appendix B — Policy DSL Grammar and Worked Examples

This appendix specifies the policy DSL as implemented in `lib/actpass/policy.ts` and walks three realistic policies through the actual evaluator. The engine is pure, deterministic, first-match-wins, and fail-closed: no LLM output enters the decision path (see §6.2).

B.1 Grammar

A policy is a JSON object validated by `validatePolicyDefinition`. Its shape:

```
{
  "id": "string",
  "version": 1,
  "description": "string (optional)",
  "applies_to": { "tools": ["..."], "agents": ["..."] },
  "mode": "monitor | warn | enforce | strict",
  "rules": [ /* PolicyRule[] */ ]
}
```

`id` (string) and `version` (finite number) are required; `mode` and `applies_to` are optional. Each `PolicyRule` is:

```
{
  "name": "string",
  "decision": "allow | deny | warn | require_approval | require_reauth | require_tool_reapproval | requi
  "when": { "all": [ /* Condition[] */ ] },
  "reason": "reason_code string",
  "approval": { "channel": "string", "min_role": "string" }
}
```

A `when` group must contain **exactly one** of `all` or `any`, and that array must be non-empty — validation rejects zero groups, both groups, or an empty array. `all` requires every condition true; `any` requires at least one. An empty or malformed group matches nothing (fail-closed).

A `Condition` is a `{ path, operator, value }` triple. `path` is a dotted accessor resolved by `resolvePath` against the normalized `PolicyContext` (e.g. `args.amount`, `passport.resource_constraints.max_amount`, `tool.name`). Any missing segment resolves to `undefined`.

Operator set (10): `==`, `!=`, `>`, `>=`, `<`, `<=`, `in`, `not_in`, `contains`, `matches`. Numeric comparisons coerce both operands with `toFiniteNumber`; when both coerce to finite numbers they compare numerically (`"1000000" > 50000` is true), otherwise they fall back to string comparison. `in` / `not_in` test membership against an array on the right; `contains` tests array membership or substring; `matches` tests a `RegExp` (an invalid regex returns false).

`$ref` semantics. A condition `value` may be a literal or a path reference `{ "$ref": "<dotted.path>" }`. When it is a `$ref`, the right-hand operand is resolved from the context instead of taken literally — enabling arg-to-passport comparisons such as `args.customer_id == passport.resource_constraints.customer_id`.

Fail-closed rules for absent inputs. A missing left operand returns false for every operator *except* `!=`, `in`, and `not_in`. `undefined != "x"` is true, so a deny-on-mismatch rule fires when a required claim is absent. `in` against a missing/non-array set is false (an allow-guard does not fire without its allowlist); `not_in` against a missing set is true (a deny-guard fires when the allowlist is absent). Every gap escalates or denies — never a silent allow.

Evaluation. `evaluatePolicy` returns `deny` + `policy.missing` for a null or out-of-scope policy, `deny` + `args.schema_invalid` for a context without `args`, walks `rules` in order returning the **first** match with its `decision`, `reason` (as `reason_code`), `matched_rules`, and optional `approval`, and if no rule matches returns `deny` + `policy.denied_default`. Every rule's condition results are captured in `evidence` for the ledger.

B.2 Example 1 — Refund threshold with an approval band

```
{
  "id": "refund_policy", "version": 3,
  "applies_to": { "tools": ["resolve_refund_request"] },
  "rules": [
    { "name": "allow_small_refund", "decision": "allow", "reason": "refund.small_in_scope",
      "when": { "all": [ { "path": "args.amount", "operator": "<=", "value": 10000 } ] } },
    { "name": "require_approval_medium_refund", "decision": "require_approval", "reason": "refund.medium
      "approval": { "channel": "slack", "min_role": "approver" },
      "when": { "all": [ { "path": "args.amount", "operator": "<=", "value": 50000 } ] } },
    { "name": "deny_large_refund", "decision": "deny", "reason": "refund.out_of_policy",
      "when": { "all": [ { "path": "args.amount", "operator": ">", "value": 50000 } ] } }
  ]
}
```

Input: `{ tool: { name: "resolve_refund_request" }, args: { amount: 25000 } }` (amount in cents). Rule 1: `25000 <= 10000` false. Rule 2: `25000 <= 50000` true → first match. **Decision:** `require_approval`, `reason_code: refund.medium_needs_approval`, `matched_rules: ["require_approval_medium_refund"]`, routing to a Slack approval requiring role `approver`. A `$1,000,000` refund arriving as the string `"100000000"` coerces numerically, skips rules 1–2, and hits `deny_large_refund` → `deny`, `refund.out_of_policy`.

B.3 Example 2 — Deploy gated on branch and CI status (enriched input)

Branch and CI status are not in the raw tool call. A per-tool enricher stamps them onto the context before preflight (see §6.6); the engine does no I/O.

```
{
  "id": "github_pr_merge_deploy", "version": 2,
  "applies_to": { "tools": ["merge_and_deploy"] },
  "rules": [
    { "name": "block_non_ci_pass", "decision": "deny", "reason": "policy.denied_by_rule",
      "when": { "all": [ { "path": "args.ci_status", "operator": "!=", "value": "passed" } ] } },
    { "name": "prod_needs_approval", "decision": "require_approval", "reason": "policy.approval_required
      "approval": { "channel": "slack", "min_role": "security_admin" },
      "when": { "all": [
        { "path": "args.target_branch", "operator": "==", "value": "main" },
        { "path": "args.ci_status", "operator": "==", "value": "passed" } ] } },
    { "name": "allow_feature", "decision": "allow", "reason": "policy.allowed",
      "when": { "any": [
        { "path": "args.target_branch", "operator": "!=", "value": "main" } ] } }
  ]
}
```

Input (enriched): `{ tool: { name: "merge_and_deploy" }, args: { target_branch: "main", ci_status: "passed" } }`. Rule 1: `"passed" != "passed"` false. Rule 2: both conditions true → first match. **Decision:** `require_approval` to `security_admin`. Now consider the failure mode: if the enricher cannot reach CI and omits `ci_status`, `args.ci_status` is `undefined`; `undefined != "passed"` is **true**, so `block_non_ci_pass` fires → `deny`, `policy.denied_by_rule`. A missing enrichment signal denies rather than deploying blind.

B.4 Example 3 — Data export with a row limit and PII guard

```
{
  "id": "data_export", "version": 4,
  "applies_to": { "tools": ["export_dataset"] },
  "rules": [
    { "name": "deny_pii_bulk", "decision": "deny", "reason": "policy.denied_by_rule",
      "when": { "all": [
        { "path": "args.includes_pii", "operator": "==", "value": true },
        { "path": "args.row_count", "operator": ">", "value": 1000 } ] } },
    { "name": "large_export_review", "decision": "require_approval", "reason": "policy.approval_required
      "approval": { "channel": "email", "min_role": "auditor" },
      "when": { "any": [
        { "path": "args.row_count", "operator": ">", "value": 10000 },
        { "path": "args.destination", "operator": "not_in",
          "value": { "$ref": "passport.resource_constraints.allowed_destinations" } } ] } },
    { "name": "allow_small", "decision": "allow", "reason": "policy.allowed",
      "when": { "all": [ { "path": "args.row_count", "operator": "<=", "value": 10000 } ] } }
  ]
}
```

Input: `{ tool: { name: "export_dataset" }, args: { includes_pii: false, row_count: 5000, destination: "s3://reports" }, passport: { resource_constraints: { allowed_destinations: ["s3://reports"] } } }`. Rule 1: `includes_pii == true` false. Rule 2 `any`: `5000 > 10000` false; `"s3://reports" not_in ["s3://reports"]` false → group false. Rule 3: `5000 <= 10000` true → first match. **Decision:** `allow`, `policy.allowed`. If the passport omitted `allowed_destinations`, the `$ref` resolves to `undefined`; `not_in` against a non-array set is true, so `large_export_review` fires → `require_approval` to an `auditor`. The destination allowlist is bound to the passport, so a request cannot export to a target the passport never authorized without human review.

> **Boundary note.** Per-time-window counters (`deploys_today`) and response-side controls (post-query row truncation, field redaction) are outside the request-time gate. The engine compares whatever the enricher injects; a stateful counter must be aggregated upstream, and response filtering is a separate enforcement point (see §11).

Appendix C — Threat-Model Enumeration

This appendix is the exhaustive enumeration behind §7. Each entry names the adversary, the attack, the ActPass control that addresses it, and the honest residual risk. Where a threat is genuinely a deployment-time or network-egress concern rather than a code control, the table says so explicitly — that distinction is load-bearing, and blurring it would misrepresent the guarantee.

The controls referenced here are documented in the pillar sections (§6.1–6.7) and the invariants published in `SECURITY.md`. The gateway-bypass rows are grounded in the living reference `docs/actpass-bypass-threat-model-2026-06-13.md`, which is itself grounded in `app/proxy/[tenant]/[server]/[...path]/route.ts`, `lib/actpass/http-forward.ts`, `lib/actpass/ssrf-guard.ts`, `lib/actpass/credential-store-db.ts`, and `lib/actpass/crypto-envelope.ts`.

C.1 In-band threats (calls that reach the gateway)

Every threat in this table is addressed by code in this repository, because the malicious action traverses the preflight pipeline (§5) before it can execute.

#	Adversary	Attack	ActPass control	Residual risk
1	Compromised prompt / poisoned tool output	Injected instruction drives a risky tool call (e.g. `refund.issue`, `github_pr_merge_deploy`)	Decision is rendered **outside the LLM**: the deterministic policy DSL (`policy.ts`, first-match-wins) evaluates the *action*, not the prose that produced it. A blocking `Decision` (`core.ts`) returns before execution.	The gate binds the action, not intent — an injected call that is itself *policy-compliant* still executes. Policy quality is the buyer's responsibility.
2	Over-permissioned agent	Excessive agency: agent attempts actions outside its role	RBAC static matrix (`rbac.ts`, 8 roles / 17 permissions) plus per-action risk tier (`RiskTier`) and policy scoping. High/critical tiers route to `require_approval`.	Scope is only as tight as the deployed policy and passport claims. A broad passport grants broad reach until it expires (TTL clamp: 900 s default, 3600 s max, `passport-ttl.ts`).
3	Network attacker / replay	Passport replay — reuse of a captured `jti` to repeat a mutation	Single-use enforcement: `acquireExecutionClaim` runs before credential binding; a `jti` replays at most once, idempotently, only with an identical request hash (`SECURITY.md` §3).	Idempotent re-drive within the same request hash is by design; a strictly-once semantic at the upstream is the upstream's concern.
4	Network attacker / insider	Passport forgery — mint or tamper with an Action Passport	EdDSA/Ed25519 JWT verified against the issuer JWK (`passport.ts`); any signature or claim mismatch fails closed.	Forgery reduces to stealing the issuer signing key — explicitly out of scope (`SECURITY.md` "Out of scope").
5	Racing agent	Approval swap / TOCTOU: alter arguments after approval, or reuse an approval for a different action	Approval is bound to the action's argument hash, re-verified server-side (no client-trusted fields, `SECURITY.md` §2); approvals FSM (`approvals-fsm.ts`) enforces legal state transitions.	A within-scope confused-deputy (same tool, semantically adjacent args) can pass if the policy does not distinguish them — tighten policy granularity.
6	Malicious tool publisher	Manifest rug-pull: a tool's behavior changes after certification	Manifest scan + drift detection (`scanner.ts`, `drift.ts`, `manifest*.ts`); a changed manifest hash breaks the drift check and invalidates the signed attestation (`report-signing.ts`, detached JWS).	Drift is detected at scan/preflight time, not continuously in-flight; a tool that mutates *between* the scan and a call within the same passport window is a gap.
7	Cross-tenant adversary	Read another tenant's approvals, evidence, credentials, or keys	Tenant is resolved **key-is-identity** from the authenticated principal, never the `:tenant` path segment; all queries are tenant-scoped; credential AAD binds ciphertext to tenant+provider so a blob cannot be transplanted between rows (`crypto-envelope.ts`, `credential-store-db.ts`).	Isolation is enforced in application queries against a shared database, not by physical tenancy — a query-construction bug is the failure mode this invariant exists to catch (`SECURITY.md` §4).
8	Insider / storage tamperer	Evidence tampering or fork of the ledger	Append-only hash-chained evidence ledger (`evidence-store.ts`); each entry chains the prior hash, so any edit or truncation breaks verification (`SECURITY.md` §5).	The chain makes tampering **detectable**, not impossible; detection depends on retaining an independent anchor (e.g. a signed export, §8) to compare against. A silent full-chain re-write with a

#	Adversary	Attack	ActPass control	Residual risk
				colluding anchor is out of scope.
9	Malicious agent	Credential exfiltration — read the vault secret through the gateway	Secrets are AES-256-GCM encrypted at rest, bound server-side on the gateway path only, and **never returned to the caller or written to evidence** (`SECURITY.md` §6).	The secret is decrypted in gateway memory at execution time; in managed hosting this is inside ActPass's trust boundary. Buyers who require the secret to never leave their network use the in-VPC self-hosted gateway.
10	Malicious upstream config / agent	SSRF via egress — point a forward at internal infrastructure	Upstream destinations are server-configured (`ACTPASS_UPSTREAM_<SERVER>`), never client-supplied; every forward is SSRF-guarded to TLS-only, non-private egress (`ssrf-guard.ts` , `http-forward.ts`).	The guard blocks private/loopback/link-local hosts; a *misconfigured public* upstream is still reachable. A boot-time `ACTPASS_UPSTREAM_*` validator is the recommended belt-and-suspenders. A known residual (flagged in-source): the guard validates by re-resolving DNS and does **not** pin the validated address at connect, so a DNS-rebind TOCTOU — a name that resolves public at check-time and private at connect-time — is not yet closed. The upgrade path is pinning the checked IP at connect.
11	Malicious or careless policy author	Poisoned policy authoring — a hostile or negligent policy change neuters the gate (e.g. a leading broad `allow` placed above the rules that were meant to block)	Policy files are parsed and structurally validated before load; the engine is deterministic and **first-match-wins** (`policy.ts`), so evaluation is predictable and auditable.	Structural validation is not intent validation — a syntactically valid policy can still be permissive by design or by accident, and first-match-wins ordering makes rule *position* a poisonable surface: one broad `allow` at the top silently neuters every rule beneath it. Policy authoring is itself a trust boundary; the mitigation is a human / second-control review gate on every policy change, not something the parser can catch.

C.2 Out-of-band threats (gateway-bypass vectors)

These vectors do not defeat the proxy — **the proxy is correct** — they route *around* it. The malicious call never reaches ActPass, so no in-repo control can see it. The enforcement claim collapses only if the agent can reach the upstream without traversing the gateway, and that precondition is a property of **how the agent is deployed**, not of this codebase. The decisive mitigation for every row below is the same: **network egress control** that permits *only* the ActPass proxy and blocks the upstream directly, converting a voluntary chokepoint into a mandatory one.

#	Adversary	Bypass vector	Mitigation (deployment-time)	Residual risk
1	Agent	Ambient credential in agent env (PAT/token in env var, `~/.config`, OS keychain) — agent calls the upstream directly	Strip all upstream creds from the agent runtime; bind only in the vault. Egress control makes it moot.	Any secret the agent can read, it can use directly absent egress control.
2	Agent	Shelling out to `git` / `gh` with local creds	Remove git creds and SSH deploy keys from the agent host	A local binary with cached creds bypasses the HTTP proxy entirely.
3	Operator error	Second / ungated MCP server carrying its own token	Inventory and gate **every** tool surface; one ungated server is a hole	Coverage is only as complete as the tool inventory.
4	Agent	Agent writes and runs a script using a readable token	Same as row 1, plus no readable secrets on disk	Indirection does not change the root cause: a readable secret.
5	Agent	SSH git remotes / agent forwarding — a non-HTTP path	Disable SSH egress from the agent host	Any non-HTTP egress path is invisible to an HTTP proxy.

Because these are empirical, coverage is **measured, not asserted**: the pilot instrumentation (§9) compares the upstream's independent audit log against ActPass `chain_id`s over real agent use, and every audited sensitive action with no matching `chain_id` is a counted bypass.

C.3 Explicit non-goals

Stated plainly so a skeptical reader is not misled about the boundary of the guarantee:

- **Reads are not gated** — for latency, read APIs are out of scope for the mutation gate. Data exfiltration through reads requires a separate response-filtering layer that ActPass does not provide.
- **Volumetric denial of service** is out of scope (`SECURITY.md`).
- **A compromised gateway host or stolen signing keys** are out of scope — every signature and confidentiality guarantee reduces to key custody, and ActPass does not defend a host the adversary already owns.
- **Third-party service security** (Stripe, Neon, Sentry) is the vendor's responsibility.
- **The gateway is not a mandatory chokepoint by itself.** Mandatoriness is supplied by the deployment's egress policy; without it, ActPass is a voluntary chokepoint and coverage is a deployment property, not a code guarantee. This is the single most important honesty in this appendix.

Appendix D — Data-Model Summary

The persistence layer is a single Drizzle schema (`lib/db/schema.ts`, 1,031 LOC) defining **34 `pgTable` tables** on Neon serverless Postgres (HTTP driver, no interactive transactions — atomic multi-statement work is expressed as plpgsql functions in migrations `0023` / `0024`). Every enforcement-bearing table carries a `team_id` foreign key to `teams`, so tenant scoping is a column, not a convention: queries filter on `team_id` and cross-tenant reads are structurally impossible without it. This appendix summarizes the security-critical subset; the SQL of record is the 25 migrations under `lib/db/migrations/`.

The organizing invariant of the model is that **integrity-bearing rows are append-only and their tamper anchors are enforced by database uniqueness constraints**, not by application code that a compromised process could skip. Where a value must be single-use (a passport `jti`, a Stripe `session_id`, an evidence chain link), the constraint lives in Postgres, so a replay or fork surfaces as a unique violation regardless of what the caller intended.

Core tables (selected from 34)

Table	Purpose	Security-critical fields / constraints
`users`	Human identity	`email` unique; `password_hash` (never plaintext); `deleted_at` for soft-delete / erasure
`teams`	Tenant boundary	`stripe_customer_id` / `stripe_subscription_id` unique; `public_trust_slug` unique, opt-in, unguessable (never the raw id)
`team_members`	RBAC binding	`(user_id, team_id)` unique index; `role` maps to the static RBAC matrix (see §6.5)
`agents`	Registered AI agent	`capabilities_json` — the declared upper bound; passport minting clamps requested tools to it; `status` (`active` / `drifted` / `pending_review`)
`agent_installations`	Per-device developer key	`developer_key_hash` unique (SHA-256 only); `developer_key_prefix` for display; `status` , `revoked_at` for surface revocation
`api_keys`	Team-wide gateway/SDK keys	`key_hash` unique (SHA-256 only); `prefix` ; `scope` (default `gateway`); `status` , `expires_at` — realizes API-key-is-identity
`action_passports`	Minted Action Passport records	`expires_at` ; `scope_json` ; `approval_hash` ; `policy_snapshot` ; `status` (default `active`)
`passport_uses`	Single-use replay defense	`jti` unique — durable across restarts/replicas; a second use of a `jti` is a unique violation
`passport_revocations`	Revocation list	`jti` unique; tenant-scoped; checked on every verify and just-before-use
`tool_manifests`	Immutable manifest registry	`manifest_hash` ; `input_schema_hash` / `output_schema_hash` ; `approval_status` (`unreviewed` / `approved` / `reapproval_required` / `blocked`); `drift_signals` — a differing later row flips the tool to reapproval
`actpass_policies`	Tenant policy definitions	`rules_json` (deterministic first-match-wins DSL); `team_id` -scoped; evaluated outside the LLM (see §6.2)
`runtime_decisions`	First-class decision ledger	`decision` , `reason_code` , `policy_hash` , `tool_manifest_hash` , `request_hash` , `chain_id` , `mode` — one immutable row per preflight, queryable for SIEM/analytics
`evidence_logs`	Hash-chained evidence event	`chain_id` , `previous_event_hash` , `sealed_event_json` (canonical pre-hash body), `cryptographic_signature` , `event_mac` (HMAC the DB principal cannot forge); unique `(chain_id, previous_event_hash)` + partial unique genesis index
`evidence_chain_anchors`	Signed chain head (truncation defense)	unique `(team_id, chain_id)` ; `length` , `tip_hash` , detached Ed25519 signature (`sig_protected` / `sig_signature` / `kid`) — deleting tail rows is detectable
`human_approvals`	Human-in-the-loop requests	`status` ; `version` (optimistic lock — every decide bumps it so concurrent FSM self-loops collide)
`approval_events`	Immutable reviewer audit	`decision` , `reviewer_user_id` , `event_hash` , `source` — append-only history per approval
`credential_vault_items`	Encrypted secret vault	`encrypted_secret` (AES-256-GCM envelope, AAD bound to tenant+provider; raw secret never stored); `rotated_at`
`credential_bindings`	Credential-use ledger	ties a `vault_item_id` to a `runtime_decision_id` + `evidence_event_id` ; `binding_hash` ; `expires_at`

Table	Purpose	Security-critical fields / constraints
`certifications`	ActPass Certified records	`id` is an unguessable UUID (public identity, not a serial); `manifest_hash` (tamper anchor); `tier`, `score`; `record_json` + `signature_json` for stateless verification; `expires_at`
`publisher_trust_keys`	Tenant trust store	`public_key_jwk` (Ed25519); unique `(team_id, publisher_identity)` — replaces self-asserted `publisherVerified` with signature verification
`memory_records`	Integrity-bound RAG/memory	`content_hash`; `encrypted_content` (AES-256-GCM, AAD to `(team, record_key)`); `provenance`; `status` (`active` / `quarantined`); unique `(team_id, record_key)`
`execution_claims` / `runtime_counters`	Concurrency + rate/breaker state	unique `(team_id, tool_name, request_hash)` mutex; unique `(team_id, counter_key, window_start)` fixed-window counter
`processed_stripe_sessions`	Checkout replay defense	`session_id` unique — a replayed success URL is a no-op

Supporting tables (`activity_logs`, `invitations`, `mcp_scans`, `actpass_reports`, `runtime_telemetry`, `install_sessions`, `entitlements`, `usage_events`, `support_tickets`) round out onboarding, telemetry, zero-trust device-authorization install, and billing.

Invariant-enforcing constraints

The constraints below are the load-bearing ones for the guarantees in §7:

- **Passport single-use:** `passport_uses.jti UNIQUE` — durable replay rejection across processes.
- **Chain non-fork:** `evidence_logs` unique `(chain_id, previous_event_hash)` plus a partial unique index on `chain_id where previous_event_hash is null` (Postgres treats NULLs as distinct, so the partial index closes the two-genesis-rows case).
- **Chain non-truncation:** `evidence_chain_anchors` unique `(team_id, chain_id)` carries a signed `length` / `tip_hash`; verification fails closed if loaded rows disagree.
- **Approval concurrency:** `human_approvals.version` optimistic lock; the `UPDATE` matches on it so racing decisions collide even on FSM self-loops.
- **Credential non-transplant:** vault and integration blobs bind AAD to `(tenant, provider)` / `(team, channel)`, so a row copied between tenants fails the GCM tag on decrypt.
- **Hashed-secret storage:** `api_keys.key_hash` and `agent_installations.developer_key_hash` store only SHA-256 (both `UNIQUE`); a database dump yields no usable key material.

Appendix E — Glossary and References

E.1 Glossary

Definitions below are scoped to the meanings ActPass gives these terms. Where a term maps to a specific pillar or section of this paper, that section is named for cross-reference.

Action Passport. A short-lived, EdDSA/Ed25519-signed JWT (see JWS) that ActPass mints for a delegated agent session or individual tool action. It binds the actor, purpose, tool, resource scope, expiry, approval state (see approval_hash), and policy anchor into one verifiable credential. Passports are audience/tenant/tool/resource-bound, carry replay and revocation state, and are published against a rotating key set (see JWKS). See §6.

Preflight. The deterministic decision the gateway renders for a proposed action *before* any upstream call. Exposed as `POST /v1/actions/preflight` (a dry-run decision) and reused inside the execute path. Preflight runs the five-gate pipeline — passport, policy, drift, identity, approval — outside the LLM and emits a `reason_code` plus HTTP status. See §5.

Evidence ledger / evidence fabric. The append-only, SHA-256 hash-chained record into which every decision is sealed. Each entry links to its predecessor, giving content-level tamper detection and completeness scoring; the ledger supports verification and export to JSON, Markdown, CSV, and SIEM formats. "Fabric" denotes the ledger together with its export, signing, and SIEM-forwarding surface. See §6 and §8.

Manifest drift. A change in a connected tool's manifest — code, schema, or descriptor meaning — relative to the manifest ActPass last recorded. Drift detection compares against a stored `manifest_hash` and can force re-consent when a tool's behavior or meaning changes (a "rug-pull"). See §6.

Fail-closed. The invariant that any indeterminate condition — network error, missing signing key, unverifiable claim, or no matching policy — resolves to *deny*, never *allow*. On a fresh workspace the default preflight response is `policy.denied_default`; that deny is the intended success signal. See §7.

API-key-is-identity. The rule that the tenant (and thus authorization context) is resolved from the authenticated principal — the presented API key — never from the request body or URL. Tenant IDs, approval hashes, and manifest hashes are never trusted from the client.

Enricher. A per-tool gateway adapter that populates the `PolicyContext` with derived inputs and stateful counters — CI status, protected-path globs, business-hours flags, running counters — *before* deterministic policy evaluation, so the policy engine decides on complete context. Enrichers fail closed: if a required enrichment is missing or cannot be resolved, the decision denies rather than proceeding on partial context. See §6.2.

approval_hash. A hash that binds a human approval to the exact payload it authorized. The approval FSM unlocks a blocked action only when the payload presented at execute time matches the approved hash, preventing a swap between the approved and executed arguments. See §6.

Developer key. A hashed, rotatable key (prefix `apdv_`) minted per installation (device) and shown once. It authenticates the calling principal and, under API-key-is-identity, resolves the tenant. Keys are RBAC-scoped. Distinct from the agent/gateway API key (prefix `apk_`), which authenticates agent and gateway traffic on the data plane.

ActPass Certified. A signed, ledger-sealed certification of an MCP server's tool manifest at gold / silver / bronze tiers, bound to the manifest's hash. It carries a public verify page and an embeddable, signed badge SVG; it flips to Drift or Expired automatically when the manifest changes or the certification lapses. See §8.

JWKS (JSON Web Key Set). The published set of public keys (RFC 7517) against which third parties verify Action Passports and signed reports without a shared secret. Served at `/v1/passports/jwks`.

JWS (JSON Web Signature). The signature envelope (RFC 7515) used for passports and for detached report/attestation/badge signing. ActPass uses EdDSA (Ed25519) and, for reports, the unencoded/detached-payload profile of RFC 7797.

AAD (Additional Authenticated Data). The authenticated-but-not-encrypted binding fed to the credential vault's AES-256-GCM envelope. ActPass binds AAD to tenant + provider so an encrypted secret cannot be transplanted to another row. See §6.

Reason code. A typed, stable machine-readable label (e.g. `policy.denied_default`, `passport.expired`) returned with every decision and recorded in evidence, so callers and auditors can branch on cause rather than parse prose. See §5.

Enforcement gateway. The neutral reference monitor that sits between the agent and its tools, mediating every risky call through preflight and, on execute, forwarding to the upstream through an SSRF-guarded, credential-binding path. See §4.

Control plane / data plane. The control plane is the management surface — policy authoring, approvals, keys, certification, evidence review. The data plane is the runtime request path — preflight, execute, proxy — where decisions are enforced and actions forwarded. See §4.

E.2 References

The paper leans on the following standards and external instruments, drawn from the sources ActPass maps its controls against. Where an instrument's exact clause numbering is version-dependent, it is cited generally rather than by article.

1. NIST, *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*, NIST AI 100-1 — the Govern / Map / Measure / Manage functions and trustworthiness characteristics ActPass maps controls to; and the *Generative AI Profile* (NIST AI 600-1).
2. Regulation (EU) 2024/1689, *the EU AI Act* — provisions on record-keeping / automatic logging, human oversight, transparency, robustness, and cybersecurity for high-risk AI systems.
3. OWASP, *Top 10 for Large Language Model Applications*, and the OWASP agentic-security guidance — prompt injection and excessive agency as primary failure modes; findings in the ActPass scanner carry OWASP LLM Top 10 mappings.
4. IETF RFC 8628, *OAuth 2.0 Device Authorization Grant* — the device-code flow underpinning ActPass browser-attested CLI pairing.
5. IETF RFC 7797, *JSON Web Signature (JWS) Unencoded Payload Option* — the detached-payload profile used for third-party-verifiable report, attestation, and badge signatures.
6. IETF RFC 7515, *JSON Web Signature (JWS)*, and RFC 7517, *JSON Web Key (JWK)* — the signature and public-key-set formats used for passports and JWKS.
7. Model Context Protocol specification and its security best-practices guidance — untrusted tool metadata, explicit user consent before tool invocation, OAuth 2.1, audience-bound tokens, and the prohibition on token passthrough.
8. IETF RFC 8032, *Edwards-Curve Digital Signature Algorithm (EdDSA)* — the Ed25519 scheme used for passport and report signatures.
9. NIST, *Special Publication 800-38D* — Galois/Counter Mode (GCM), the authenticated-encryption mode used by the AES-256-GCM credential vault.
10. NIST, *Special Publication 800-207, Zero Trust Architecture* — least-privilege, per-session access evaluated under dynamic policy, the action-time model ActPass implements.
11. ISO/IEC 42001, *AI management systems* — the management-system wrapper for AI governance and continuous improvement referenced in the standards mapping (§8).
12. OASIS / open policy-as-code references — Open Policy Agent and the Cedar policy language, cited as externalized authorization prior art for the deterministic policy DSL (§6).