Agent Control Protocol

ACP v1.19: Admission Control for Agent Actions

Technical Specification and Reference Implementation

Marcelo Fernandez
TraslaIA
info@traslaia.com
https://agentcontrolprotocol.xyz

March 2026
Draft Standard
arXiv:2603.18829 [cs.CR] DOI: 10.5281/zenodo.19219776

Abstract

ACP defines a deterministic admission control protocol for autonomous agents, combining capability, resource, and temporal signals (history, anomaly, cooldown) into runtime admission decisions that existing stateless frameworks (RBAC, ABAC, OPA, Cedar) cannot enforce. The key design insight is that the protocol is **compute-cheap but state-sensitive**: decision evaluation is computationally inexpensive (∼820 ns), while system performance is primarily constrained by state access patterns—a separation that allows the state backend to be replaced or scaled without modifying protocol semantics. Experimental evaluation demonstrates throughput up to ∼920k req/s under moderate concurrency, with the cooldown containment path at ∼88 ns (9.5× faster via Step 2 early exit), and throughput degrading gracefully to ∼712k req/s at 500 workers due to state backend contention. Adversarial evaluation across three attack scenarios—cooldown evasion, distributed multi-agent coordination, and state backend stress—confirms that ACP-RISK-2.0 enforcement holds under active evasion: 99% (495/500) of single-agent evasion attempts are blocked after only five requests (Experiment 1), per-agent isolation is preserved across 100 coordinated agents, and throughput degradation under stress is attributable solely to state-backend latency rather than decision logic. ACP provides end-to-end verifiability through 73 signed conformance test vectors, a TLC-checked TLA+ formal model (3 invariants, 0 violations), and an ACR-1.0 sequence compliance runner that validates stateful multi-step behavior at runtime.

Specification and implementation: https://github.com/chelof100/acp-framework-en

Contents

1 The Problem ACP Solves

Autonomous agents are being deployed in institutional environments without a technical standard to govern their behavior. This is not a tooling problem—it is a protocol problem.

1.1 The Structural Gap

When an autonomous agent makes a decision and executes it, there is a critical moment between both actions. In current models, that moment does not formally exist: the decision and the execution are the same event. The agent decides and acts. There is no intermediate validation. There is no point of intervention. There is no structured record of why the decision was made.

This is acceptable when a human executes that action, because the human bears responsibility and can be questioned. An autonomous agent cannot be questioned. It can only be audited—and only if there is something to audit.

> *The problem is not whether agents are trustworthy. The problem is that currently no formal technical mechanism exists that allows an institution to demonstrate that its agents operated within authorized limits.*

1.2 ACP as Admission Control

The clearest frame for understanding what ACP does is the Kubernetes Admission Controller analogy.

Kubernetes intercepts every API request before it reaches the cluster and runs it through a sequence of admission checks—`ValidatingWebhookConfiguration`, `ResourceQuota` enforcement, OPA Gatekeeper policies. If any check fails, the request is rejected before touching cluster state.

ACP applies this pattern to agent actions:

```
agent intent
    |
[1] Identity check     (ACP-AGENT-1.0, ACP-HP-1.0)
    |    is this agent who they claim to be?
[2] Capability check   (ACP-CT-1.0, ACP-DCMA-1.1)
    |    does the agent hold a valid token for this action?
[3] Policy check       (ACP-RISK-2.0, ACP-PSN-1.0)
    |    is this action within current policy?
[4] ADMIT / DENY / ESCALATE
    |    (if ADMIT)
[5] Execution token    (ACP-EXEC-1.0)
    |    single-use cryptographic proof of admission
[6] Ledger record      (ACP-LEDGER-1.3)
    |    immutable signed audit entry
system state mutation
```

The critical difference from Kubernetes: ACP's admission check operates across institutional boundaries. An agent from Bank A can be admitted by Bank B without Bank B trusting Bank A's

internal infrastructure—the cryptographic proof is self-contained and verifiable with Bank A's published public key alone.

This *admission control* framing also clarifies the relationship with related tools:

- **OPA (Open Policy Agent)** [10] can serve as the policy evaluation engine inside Step 3. ACP does not replace OPA; it adds the identity and delegation chain layers above it.
- **AWS IAM / Azure RBAC** model static role permissions for humans. ACP adds dynamic agent delegation with execution proof.
- **OAuth 2.0** [5] handles API access tokens. ACP extends delegation to multi-agent chains with non-escalation and verifiable provenance.
- **SPIFFE / SPIRE** [14] provides cryptographic workload identity. ACP builds on that identity to add capability scoping and governance.

1.3 Why RBAC and Zero Trust Are Insufficient

RBAC and Zero Trust are the predominant control layers in enterprise environments. Both are necessary. Neither solves the problem of governing autonomous agents:

Criterion	RBAC	Zero Trust	ACP
Designed for	Human roles	Network access	Autonomous agents
Cryptographic identity	No	Partial	Yes (Ed25519, mandatory)
Verifiable dynamic delegation	No	No	Yes (chained, auditable)
Decision/execution separation	No	No	Yes (Execution Tokens)
Real-time risk evaluation	No	Partial	Yes (deterministic)
Multi-institutional auditing	Non-standard	Non-standard	Native (signed ledger)
Transitive delegation revocation	No	No	Yes (formal propagation)
B2B interoperability for agents	Unstructured	Unstructured	Central protocol design

Table 1: Comparison of ACP with RBAC and Zero Trust across agent governance criteria.

ACP does not replace RBAC or Zero Trust. It adds a governance layer oriented specifically to autonomous agents that operates above existing controls.

1.4 The Concrete Scenario ACP Prevents

Without ACP: A financial processing agent receives instructions from another agent to execute a transfer. The agent executes the action. If the instruction was legitimate, everything works. If it was compromised, injected, or generated by an unauthorized agent, the transfer occurs anyway. No formal mechanism exists to prevent it, nor is there a technical record that allows reconstructing the authorization chain.

With ACP: The agent requesting the transfer must present a Capability Token cryptographically signed by the issuing institution, demonstrate possession of the associated private key, and the request passes through the risk engine before receiving an Execution Token. The ET is single-use. The entire chain is recorded in the Audit Ledger with an externally verifiable institutional signature.

1.5 Design Insight

ACP is **compute-cheap but state-sensitive**: decision evaluation is computationally inexpensive, while system performance is primarily constrained by state access patterns.

This separation enables a protocol that is both efficient in isolation (∼820 ns per decision, ∼88 ns for the cooldown short-circuit path) and adaptable in deployment: performance can be improved by replacing the state backend without modifying protocol semantics. The `LedgerQuerier` interface formalizes this boundary—the decision function remains stateless and deterministic; the state backend can be a local in-memory structure for development or a distributed, indexed store in production.

1.6 Contributions

This paper makes the following contributions:

- **Admission control protocol.** A deterministic admission control protocol for autonomous agents combining capability, resource, historical, and anomaly-based signals into a single verifiable decision.
- **State-aware execution model.** An execution model that explicitly separates decision logic from state management via the `LedgerQuerier` abstraction, enabling deterministic reasoning about temporal agent behavior.
- **Runtime containment.** A cooldown-based containment mechanism for repeated or adversarial behavior, providing a 9.5× faster denial path (88 ns) via Step 2 short-circuit before risk score computation.
- **End-to-end verifiability.** A three-layer verifiability pipeline: formal (TLA+, TLC-checked invariants), data (signed conformance test vectors), and runtime (ACR-1.0 compliance runner), enabling independent validation without proprietary infrastructure.
- **Empirical evaluation.** Benchmarks demonstrating sub-microsecond decision latency (∼820 ns), throughput up to 920,000 requests per second, and a 6.7× throughput difference between single-agent and multi-agent workloads that identifies state backend contention as the primary performance boundary.

2 What ACP Is

A formal technical specification—not a framework, not a platform, not a set of best practices. A protocol with precise definitions, formal state models, verifiable flows, and explicit conformance requirements.

2.1 Definition

Agent Control Protocol (ACP) is a technical specification that defines the mechanisms by which autonomous institutional agents are identified, authorized, monitored, and governed in B2B environments. It establishes the formal contract between an agent, the institution that operates it, and the institutions with which it interacts.

Core invariant:

$$\mathit{Execute}(\text{request}) \implies \mathit{ValidIdentity}(\text{agent}) \land \mathit{ValidCapability} \land \mathit{ValidDelegationChain} \land \mathit{AcceptableRisk}$$

No action of an ACP agent can be executed without all four predicates being simultaneously true. If any fails, the action is denied. No exceptions.

2.2 Design Principles

ACP was designed with five principles that are non-negotiable at implementation time:

- **P1 Fail Closed.** On any internal component failure, the action is denied. Never approved by default.
- **P2 Identity is cryptography.** AgentID = base58(SHA-256(public_key)). No usernames. No arbitrary IDs. Identity cannot be claimed—it must be demonstrated in every request.
- **P3 Delegation does not expand privileges.** The delegated agent's permissions are always a strict subset of the delegator's permissions. This property is cryptographically verified at every chain hop.
- **P4 Complete auditability.** Every decision—approved, denied, or escalated—is recorded in an append-only ledger with institutional signature. Not just successes. Everything.
- **P5 External verification possible.** Any institution can verify ACP artifacts from another institution using only the public key registered in the ITA. No dependency on proprietary systems.

2.3 Formal Agent Model

In ACP, an agent is a formal tuple with well-defined state:

$$A = (\,\text{AgentID},\ \text{capabilities},\ \text{autonomy\_level},\ \text{state},\ \text{limits}\,)$$

Field	Type	Description
AgentID	String (43–44 chars)	`base58(SHA-256(pk))`. Derived from public key. Immutable.
capabilities	List of strings	Explicit permissions. Format: `acp:cap:<domain>.<action>`. Never abstract roles.
autonomy_level	Integer 0–4	Determines risk evaluation thresholds. 0 = no autonomy. 4 = maximum.
state	Enum	`active | restricted | suspended | revoked`. Transition to `revoked` is unidirectional.
limits	Object	Rate limits, maximum amounts, time windows. Not modifiable at runtime.

Table 2: ACP formal agent tuple fields.

2.4 Layered Architecture

ACP does not replace existing security infrastructure. It is added as an upper layer:

```
ACP Layer   -- Autonomous agent governance: identity, authorization, risk, auditing
RBAC Layer  -- Role-based access control for human users
Zero Trust  -- Continuous identity and network access verification
```

3 Technical Mechanisms

ACP defines six interdependent mechanisms. Each has its own formal specification, state model, data structure, protocol flow, and error codes.

3.1 Serialization and Signing (ACP-SIGN-1.0)

Every verification in ACP begins with signature verification. ACP-SIGN-1.0 defines the exact process that produces a binary result—valid or invalid—without ambiguity:

1. **Canonicalization with JCS (RFC 8785) [11].** Produces a deterministic representation of the JSON object, independent of field order and the system that generated it.
2. **SHA-256 hash** over the canonical output in UTF-8.
3. **Ed25519 signature (RFC 8032)** [7] over the hash. 32-byte key, 64-byte signature.
4. **Base64url encoding** without padding for transmission.

Signature verification precedes all semantic validation. An object with an invalid signature is rejected without processing its content. This rule has no exceptions (PROHIB-003, PROHIB-012).

3.2 Capability Token (ACP-CT-1.0)

The Capability Token is ACP's central artifact. It is a signed JSON object that specifies exactly what an agent can do, on what resource, for how long, and whether it can delegate that capability to other agents.

```
{
  "ver": "1.0",
  "iss": "<AgentID_issuer>",
  "sub": "<AgentID_subject>",
  "cap": ["acp:cap:financial.payment"],
  "res": "org.example/accounts/ACC-001",
  "exp": 1718923600,
  "nonce": "<128bit_CSPRNG_base64url>",
  "deleg": { "allowed": true, "max_depth": 2 },
  "parent_hash": null,
  "sig": "<Ed25519_base64url>"
}
```

Critical fields: `exp` is mandatory—a token without expiry is invalid by definition. The 128-bit nonce prevents replay attacks. `parent_hash` chains delegated tokens in a verifiable way. The signature covers all fields except `sig`.

3.3 Handshake and Proof-of-Possession (ACP-HP-1.0)

Possessing a valid Capability Token is not sufficient to act. ACP-HP-1.0 requires that the bearer demonstrate in every request that they possess the private key corresponding to the AgentID declared in the token. This eliminates the possibility of impersonating an agent with a stolen token.

The protocol is stateless—it does not establish sessions, does not produce a `session_id`, does not require server-side state between requests. The proof occurs in every interaction:

1. The receiving system issues a 128-bit challenge generated by CSPRNG, valid for 30 seconds and single-use.
2. The agent signs the challenge together with the HTTP method, path, and body hash of the request.
3. The receiver verifies the signature using the agent's public key, obtained from the ITA.
4. The challenge is deleted immediately after use—it cannot be reused.

This sequence guarantees four formal properties: identity authentication, cryptographic request binding, anti-replay, and transport channel independence.

3.4 Deterministic Risk Evaluation (ACP-RISK-2.0)

Each authorization request passes through a deterministic risk function that produces a Risk Score (RS) in the range $[0, 100]$. The same input always produces the same result—no stochastic elements, no machine learning in the critical path. ACP deliberately trades expressiveness for deterministic auditability and reproducibility: every decision can be reconstructed from first principles given only the input request and the current state snapshot.

$$\mathrm{RS} = \min\big(100,\ B(c) + F_{\mathrm{res}}(r) + F_{\mathrm{ctx}}(x) + F_{\mathrm{hist}}(h) + F_{\mathrm{anom}}(q)\big)$$

where $F_{\mathrm{anom}}(q)$ is the anomaly factor introduced in v2.0, evaluated against the agent's accumulated request history q (pattern frequency, denial rate, cooldown state). When an agent accumulates three DENIED decisions within ten minutes, a cooldown is activated and subsequent requests are blocked regardless of RS.

The RS determines the decision according to the thresholds configured for the agent's `autonomy_level`. With `autonomy_level` 2 (standard): $\mathrm{RS} \leq 39 \rightarrow$ APPROVED; $\mathrm{RS} \in [40, 69] \rightarrow$ ESCALATED; $\mathrm{RS} \geq 70 \rightarrow$ DENIED. An agent with `autonomy_level` 0 always receives DENIED.

3.5 Verifiable Chained Delegation (ACP-DCMA-1.1)

ACP allows an agent to delegate capabilities to another agent, which in turn can delegate to a third, up to the maximum depth defined in the root token. Delegation guarantees three properties:

Factor	Description	Example values
$B(c)$	Baseline by capability	`*.read = 0`, `financial.payment = 35`
$F_{\text{ctx}}(x)$	Request context	Non-corporate IP +20; outside hours +15
$F_{\text{hist}}(h)$	Agent history (24h)	Recent denial +20; anomalous frequency +15
$F_{\text{res}}(r)$	Resource classification	`public = 0`; `sensitive = 15`; `restricted = 45`

Table 3: ACP-RISK-2.0 risk scoring factors (base formula; F_{anom} adds up to +30 from pattern accumulation).

- **No privilege escalation.** The delegated agent's capability set is always $\subseteq$ the delegator's set. Cryptographically verified at every hop via the `parent_hash` field.
- **Bounded depth.** The `max_depth` field establishes the chain limit. A chain that exceeds it is invalid.
- **Transitive revocation.** Revoking an agent's token automatically invalidates all delegated tokens that descend from it.

3.6 Execution Token (ACP-EXEC-1.0)

The separation between authorization and execution is a core principle of ACP. When the authorization engine approves a request, it returns an Execution Token (ET): a single-use artifact with a short lifetime that authorizes exactly that action, on that resource, at that moment.

- An ET can only be consumed once. A second presentation is rejected (PROHIB-002).
- An expired ET is invalid even if it was never used.
- The target system that consumes the ET notifies the ACP endpoint, closing the audit cycle.

3.7 Audit Ledger (ACP-LEDGER-1.3)

The Audit Ledger is a chain of cryptographically signed events where each event includes the hash of the previous event:

$$h_n = \text{SHA-256}(\, e_n \parallel h_{n-1} \,)$$

This structure makes it impossible to modify or delete an event without invalidating the entire subsequent chain. The ledger records all lifecycle event types: GENESIS, AUTHORIZATION (including DENIED and ESCALATED), RISK_EVALUATION, TOKEN_ISSUED, TOKEN_REVOKED, EXECUTION_TOKEN_ISSUED, and EXECUTION_TOKEN_CONSUMED.

Institutions with FULL conformance expose the ledger via `GET /acp/v1/audit/query`, allowing external partners to verify chain integrity using only the institutional ITA public key. Modifying or deleting ledger events is prohibited (PROHIB-007, PROHIB-008).

It is important to note that the ledger provides **verifiable evidence** of execution, not enforcement itself. Enforcement in ACP occurs at the policy evaluation and execution layers (ACP-EXEC-1.0, ACP-DCMA-1.1): the Execution Token is the cryptographic artifact that gates actual system-state mutation. The ledger's role is to make every admission decision—and its outcome—auditable and tamper-evident after the fact.

4 Inter-Institutional Trust

In a B2B environment, agents of one institution interact with another institution's systems. ACP defines the exact mechanism by which this trust is established, verified, and can be revoked.

4.1 Institutional Trust Anchor (ACP-ITA-1.0)

The ITA is the authoritative registry that links an `institution_id` to an Ed25519 public key. It is the only point where ACP depends on an out-of-band mechanism: the initial distribution of the ITA authority's public key. Once that key is resolved, all subsequent verification is autonomous and cryptographic.

Each institution registers in the ITA its Root Institutional Key (RIK)—the private key held in HSM that never leaves it. All ACP artifacts from that institution (tokens, ledger events, API responses) are signed with that key. Any third party can verify them by resolving the public key from the ITA.

4.2 Mutual Recognition (ACP-ITA-1.1)

When two institutions operate under different ITA authorities, ACP-ITA-1.1 defines the mutual recognition protocol. The process requires both authorities to sign a Mutual Recognition Agreement (MRA) establishing: the scope of recognition, the agreement's validity period, and the proxy resolution mechanism.

Recognition is explicitly non-transitive. If A recognizes B and B recognizes C, A does not automatically recognize C. Each bilateral relationship requires its own signed MRA. This prevents uncontrolled expansion of the trust graph.

4.3 Institutional Key Rotation and Revocation

Normal rotation includes a transition period of up to 7 days during which both keys are valid, allowing artifacts signed with the old key to be verified during the transition.

Emergency rotation is activated when a key is compromised. The result is immediate: the key is marked `revoked`, all artifacts signed with it are invalid from that moment, and there is no transition period.

5 Cross-Organization Execution

ACP-CROSS-ORG-1.1 defines a fault-tolerant bilateral protocol for interactions between independent institutions, each operating its own ACP ledger. The protocol closes five gaps identified in

the 1.0 version: undefined status tracking, absent retry protocol, unregistered ACK event type, `pending_review` without SLA, and a stale header reference.

5.1 Interaction Model

A cross-organization interaction involves two ACP-compliant institutions: a **source institution** that originates the bundle and a **target institution** that validates, executes, and acknowledges it. The protocol operates asynchronously—the source does not block waiting for an ACK.

Each interaction carries a mandatory `interaction_id` (UUIDv7), immutable and reused across all retries. This is distinct from `event_id` (UUIDv4), which is unique per emission. The target deduplicates by `interaction_id`, not by `event_id`.

For example, an agent in Institution A may request execution in Institution B. The request is recorded as a `CROSS_ORG_INTERACTION` event in A's ledger, transmitted to B, validated against B's policy and capability registry, and resolved through a `CROSS_ORG_ACK` event registered in both ledgers. Institution A derives final execution state from the ACK without storing mutable status.

5.2 Fault Tolerance and Retry Protocol

When no `CROSS_ORG_ACK` is received within the 300-second timeout, the source retries up to three times with exponential backoff:

Attempt	Wait after timeout
1 (initial)	—
2	+30 s
3	+60 s
4 (final)	+120 s

After three failed attempts the interaction transitions to `retry_exhausted` (CROSS-012) and an operational alert is raised. If a valid ACK arrives at any point, all retry timers are cancelled immediately.

5.3 Derived Interaction Status

The status of a cross-organization interaction is never stored as mutable state. It is always derived from events present in the ledger:

Derived status	Condition
`pending_ack`	`CROSS_ORG_INTERACTION` exists, no ACK
`acked`	ACK with `status = accepted`
`rejected`	ACK with `status = rejected`
`pending_review`	ACK with `status = pending_review`
`expired`	`pending_review` AND `now > review_deadline`

Precedence rule: `accepted` > `rejected` > `pending_review`. A mutable `CROSS_ORG_STATUS_UPDATE` event is explicitly prohibited as an anti-pattern that introduces race conditions and divergent audit trails.

5.4 Interaction State Invariants

For any `interaction_id`, ACP enforces the following invariants:

- **At most one terminal state.** Only one `accepted` or `rejected` ACK may exist per `interaction_id`; a duplicate with a different payload is rejected (CROSS-014).
- **ACK dominates retries.** A valid `CROSS_ORG_ACK` cancels all pending retry timers immediately, regardless of attempt number.
- **Immutable correlation key.** Retry operations MUST NOT create new `interaction_id` values; only `event_id` changes across attempts.
- **No terminal-to-non-terminal regression.** Once `accepted` or `rejected`, the status cannot change. The transition `pending_review` → `pending_review` is also prohibited (CROSS-015).

5.5 Pending Review SLA

When the target returns `pending_review`, a 24-hour SLA applies. The ACK payload includes a `review_deadline` field (Unix seconds). Valid terminal transitions are: `accepted`, `rejected`, or `expired` (implicit, when `now > review_deadline`).

5.6 CROSS_ORG_ACK as a First-Class Ledger Event

`CROSS_ORG_ACK` is registered in ACP-LEDGER-1.3 §5.15 as a first-class event type, signed with Ed25519 and serialized via JCS (RFC 8785). This enables verifiable and auditable execution across independent institutional boundaries, eliminating the audit gap that existed in version 1.0.

5.7 Security Considerations

ACP-CROSS-ORG-1.1 assumes authenticated communication channels between institutions. All `CROSS_ORG_INTERACTION` and `CROSS_ORG_ACK` events are signed with the originating institution's Ed25519 ITA key and serialized via JCS (RFC 8785), providing authenticity, integrity, and non-repudiation.

Without this cryptographic layer, cross-organization events would be subject to forgery, replay, and unauthorized status derivation. Specifically, ACP mitigates:

- **Replay attacks:** `interaction_id` (UUIDv7) and `event_id` (UUIDv4) allow the target to detect and reject duplicate submissions.
- **Event forgery:** Ed25519 signatures over JCS-canonical payloads prevent an adversary from injecting valid-looking ACK or interaction events.
- **Audit divergence:** Because `CROSS_ORG_ACK` is a first-class ledger event on both sides, independent auditors can verify consistency without trusting either institution's mutable state.

ACP does not eliminate all sources of failure in distributed systems, but provides a structured model for detecting, handling, and auditing them.

6 Reputation Snapshot Portability (ACP-REP-PORTABILITY-1.1)

ACP-REP-PORTABILITY-1.1 defines the `ReputationSnapshot`: a compact, cryptographically signed record that carries an agent's reputation score across organizational boundaries. Unlike the bilateral attestation protocol of v1.0, this specification focuses on the snapshot object itself—its structure, signing procedure, validation algorithm, and expiration semantics—enabling any verifier to independently validate a snapshot without trusting an intermediary.

6.1 ReputationSnapshot Structure

A `ReputationSnapshot` contains ten fields: `ver` (version), `rep_id` (UUID v4), `subject_id`, `issuer`, `score` (float), `scale` (`"0-1"` or `"0-100"`), `model_id`, `evaluated_at` (Unix seconds), `valid_until` (Unix seconds), and `signature` (Ed25519, base64url). Fields `scale`, `model_id`, and `valid_until` are new in v1.1; v1.0 snapshots omit them and are accepted without expiration enforcement (backward compatibility, §12).

6.2 Signing Procedure

The signature covers all fields except `signature` itself, serialized via JCS (RFC 8785) canonicalization, SHA-256 digest, and Ed25519 signing:

```
1. raw = json.Marshal(signableReputation)
2. canonical = jcs.Transform(raw)                    // RFC 8785
3. digest = SHA-256(canonical)
4. sig = Ed25519.Sign(privKey, digest)
5. snapshot.signature = base64url(sig)               // no padding
```

JCS canonicalization is mandatory. Using `json.Marshal` directly is prohibited: key ordering differs across Go, Python, and TypeScript implementations, producing verification failures in cross-org deployments.

6.3 Validation Invariants

The validator enforces five invariants (REP-001 through REP-011): (1) `evaluated_at` $\leq$ `valid_until` prevents retroactively backdated snapshots; (2) `now` $\leq$ `valid_until` enforces expiration (v1.1 only, error REP-011); (3) score within `scale` bounds (REP-002); (4) non-empty `issuer` (REP-004); (5) valid Ed25519 signature (REP-010).

Structural validation (`Validate`) and cryptographic verification (`VerifySig`) are intentionally separate operations, enabling lightweight ingestion-time checks without requiring the issuer's public key.

6.4 Divergence Semantics

When a verifier receives snapshots of the same `subject_id` from multiple issuers, it may compute divergence: $|a.\text{score} - b.\text{score}|$. If the divergence exceeds a configurable threshold (default 0.30 for `scale="0-1"`), the verifier emits warning REP-WARN-002. This is non-blocking—the policy decision of whether to accept, escalate, or reject remains with the verifier's business logic, preserving institutional sovereignty.

7 Multi-Organization Interoperability Demo (GAP-14)

The `examples/multi-org-demo/` directory provides an executable end-to-end demonstration of cross-organizational ACP exchange, combining ACP-POLICY-CTX-1.1 and ACP-REP-PORTABILITY-1.1 in a single runnable scenario deployable with Docker in under five minutes.

7.1 Scenario

Two independent organizations, Org-A and Org-B, interact over HTTP. Org-A is the *issuer*: it evaluates an agent request against its local policy and produces two signed artifacts—a `PolicyContextSnapshot` and a `ReputationSnapshot`—using its Ed25519 institutional key. Org-B is the *verifier*: it independently validates both artifacts using the `pkg/policyctx` and `pkg/reputation` packages (no logic duplicated) and renders an autonomous admission decision.

The validation sequence in Org-B is:

1. `policyctx.VerifySig(pcs, pubKey)` — verifies institutional signature (ACP-POLICY-CTX-1.1 §6 step 12).
2. `policyctx.VerifyCaptureFreshness(pcs, 300s)` — enforces $\delta_{\max}$ freshness invariant.
3. `reputation.Validate(rep, now)` — structural invariants REP-001 through REP-011.
4. `reputation.VerifySig(rep, pubKey)` — Ed25519/JCS/SHA-256 cryptographic verification.
5. `reputation.CheckDivergence(orgA, orgB, 0.30)` — divergence check against Org-B's own score.
6. Final ACCEPT / DENY decision under Org-B's sovereign policy.

7.2 Divergence Handling and Institutional Sovereignty

If Org-B holds its own score for the same `subject_id`, it computes divergence $|a.\text{score} - b.\text{score}|$. If this exceeds 0.30, Org-B emits REP-WARN-002. REP-WARN-002 is non-blocking—the final decision is Org-B's prerogative, independent of Org-A's assessment.

This embodies the ACP invariant: *ACP reports divergence. ACP does not resolve divergence.* Org-B cannot extend Org-A's `valid_until`, override Org-A's `delta_max`, or substitute its own snapshot for Org-A's signed record. Each institution's attestations are cryptographically bound to its own key and immutable after issuance.

7.3 Deployment

The demo ships with a single `Dockerfile` (parameterized by `ARG ORG=org-a|org-b`), a `docker-compose.yml` with health check, and a dedicated Go module with a `replace` directive pointing to `impl/go/`:

```
docker compose up --build    # from examples/multi-org-demo/
curl http://localhost:8081/request | jq .
```

8 Threat Model

ACP explicitly defines which threats it mitigates, which properties it guarantees, and which risks fall outside its scope. For the formal adversary model and security properties, see Section 16.

8.1 STRIDE Analysis

Category	Threat	Mitigation in ACP
Spoofing	AgentID impersonation	AgentID = SHA-256(pk). Without valid signature → immediate DENIED.
Tampering	Token or event alteration	Ed25519 covers all fields. Chained ledger: altering one event invalidates all subsequent.
Repudiation	Agent denies executed action	ActionRequest digitally signed. Non-repudiation by design.
Info Disclosure	Capability exposure	Tokens reveal only the necessary subset. Channel confidentiality via TLS.
DoS	Request flooding	Rate limits per `agent_id`. Anomalous frequency control.
Elevation	Delegation that expands privileges	$C_{\text{delegated}} \subseteq C_{\text{original}}$. Cryptographically verified at each hop.

Table 4: STRIDE threat model and ACP mitigations.

8.2 Guaranteed Security Properties

ACP guarantees the following properties when the implementation is compliant:

- **Artifact integrity.** EUF-CMA security of Ed25519. Impossible to modify a token or event without invalidating the signature.
- **Identity authenticity.** Only whoever possesses `sk` can generate a valid signature under the corresponding `pk`.
- **No privilege escalation via delegation.** Demonstrable by induction over the delegation chain.
- **Anti-replay.** The single-use challenge makes reusing a proof of possession ineffective.
- **Effective revocation.** $\mathit{Valid}(t) = \mathit{valid\_sig} \wedge \neg \mathit{expired} \wedge \neg \mathit{revoked} \wedge \mathit{valid\_delegation}$. All four conditions must be true simultaneously.

8.3 Declared Residual Risks

ACP explicitly declares what it cannot resolve:

- **Total compromise of the RIK held in HSM.** ACP defines the emergency rotation process but cannot prevent a physical compromise of the custody infrastructure.
- **Coordinated institutional collusion.** If multiple institutions act maliciously in a coordinated manner, they can generate valid artifacts. ACP guarantees traceability, not prevention of malicious agreements between parties.
- **Implementation failures.** ACP is a specification. An implementation that violates prohibited behaviors can compromise all protocol guarantees. Conformance requires formal testing.
- **ITA bootstrap.** The only point that depends on an out-of-band channel. Once the ITA root key is resolved, everything subsequent is autonomous.

9 Conformance and Interoperability

9.1 Conformance Levels

Level	Required documents	Enabled capability
L1 — CORE	ACP-SIGN-1.0, ACP-CT-1.0, ACP-HP-1.0	Token issuance with cryptographic P
L2 — SECURITY	L1 + ACP-RISK-2.0, ACP-REV-1.0, ACP-ITA-1.0	Risk eval (F_anom + Cooldown), tr
L3 — FULL	L2 + ACP-API-1.0, ACP-EXEC-1.0, ACP-LEDGER-1.3	Complete verifiable auditing
L4 — EXTENDED	L3 + ACP-PAY-1.0, ACP-NOTIFY-1.0, ACP-DISC-1.0, ...	Full governance suite
L5 — DECENTRAL.	L4 + ACP-D suite	Federation without central ITA

Table 5: ACP-CONF-1.2 conformance levels.

9.2 Conformance Declaration

Every compliant implementation MUST expose a public endpoint without authentication:

```
GET https://<endpoint>/acp/v1/conformance
```

This endpoint returns the achieved level, implemented documents, declared extensions, and declaration date. It allows any external partner to verify the conformance level of a counterparty before establishing an ACP relationship.

9.3 Prohibited Behaviors

ACP defines 12 behaviors that no compliant implementation can exhibit. If any is exhibited, the implementation cannot declare conformance at any level:

Code	Prohibited behavior
PROHIB-001	Approving a request when any evaluation component fails
PROHIB-002	Reusing an already-consumed Execution Token
PROHIB-003	Omitting signature verification on any incoming artifact
PROHIB-004	Treating a not-found `token_id` as active in a revocation context
PROHIB-005	Allowing state transition from `revoked`
PROHIB-006	Issuing an ET without a prior APPROVED AuthorizationDecision
PROHIB-007	Modifying or deleting Audit Ledger events
PROHIB-008	Silencing ledger corruption detection
PROHIB-009	Ignoring `max_depth` in delegation chains
PROHIB-010	Implementing an offline policy more permissive than ACP-REV-1.0
PROHIB-011	Approving requests from agents with `autonomy_level` 0
PROHIB-012	Continuing to process an artifact with an invalid signature

Table 6: ACP prohibited behaviors. Exhibiting any disqualifies conformance at all levels.

9.4 B2B Interoperability Conditions

- **L1 Interoperability:** Institution A can verify tokens from B if both implement ACP-CONF-L1, A has access to B's public key (via ITA or out-of-band), and B's tokens use ACP-SIGN-1.0 algorithms.
- **L2 Interoperability:** A can delegate to B's agents if both implement ACP-CONF-L2, are registered in a common ITA or with mutual recognition, and B's revocation endpoint is accessible to A.
- **L3 Interoperability:** A can audit B's ledger if B implements ACP-CONF-L3, A can resolve B's public key via ITA, and B exposes `GET /acp/v1/audit/query`.

10 Use Cases

ACP is sector-agnostic. The mechanisms are identical regardless of industry. What varies is the configuration of capabilities, resources, and autonomy levels.

10.1 Financial Sector — Inter-Institutional Payment Agents

ACP-PAY-1.0 extends the capability registry with formal specifications for `acp:cap:financial.payment` and `acp:cap:financial.transfer`. Mandatory constraints in the token include `max_amount` and `currency`. 12 specific validation steps cover limit verification, beneficiary validation, and time window control. In a financial B2B scenario, Bank A can authorize an agent to execute payments up to a defined amount to pre-approved beneficiaries, with a complete record verifiable by Bank B without needing shared proprietary systems.

10.2 Digital Government — Document Processing

Government agents that process documents can operate under ACP with `autonomy_level` 1 or 2, requiring human review for any action with a Risk Score above the configured threshold. Ledger traceability provides forensic evidence for regulatory audits and transparency processes.

10.3 Enterprise AI — Multi-Company Orchestration

In agent pipelines that cross organizational boundaries, ACP allows each organization to maintain formal control over what other organizations' agents can do in their systems. Chained delegation allows an agent in company A to operate in company B's systems with explicitly delegated capabilities, without B needing to trust A's internal controls—only the chain of signed tokens.

10.4 Critical Infrastructure — Monitoring and Actuation Agents

For systems where an incorrect action has irreversible consequences, `autonomy_level` 0 ensures any actuation request is DENIED without evaluating the Risk Score. The ledger provides the forensic record needed for post-incident analysis.

11 Specification and Implementation Status

ACP v1.19 is a complete Draft Standard specification with a full Go reference implementation and external verifiability through a TLC-checked formal model and an ACR-1.0 sequence compliance runner.

11.1 Active Specifications — v1.19 (38 documents)

11.2 Reference Implementation — 23 Go Packages + Demos

The Go reference implementation in `impl/go/` covers all L1–L4 conformance levels. All 23 packages pass `go test ./...`. A Python SDK (`impl/python/`) covers the ACP-HP-1.0 handshake and all ACP-API-1.0 endpoints. The `examples/multi-org-demo/` directory (GAP-14) provides a runnable two-organization scenario using the real `pkg/policyctx` and `pkg/reputation` packages; see Section 7. The `examples/payment-agent/` directory (Sprint E) provides an executable HTTP server demonstrating the full ACP-RISK-2.0 pipeline: `POST /admission` evaluates the request with the deterministic risk engine, appends an immutable ledger event with the full factor breakdown, and triggers cooldown automatically after three DENIED decisions in ten minutes.

11.3 Conformance Test Vectors — 73 Signed Vectors

The `compliance/test-vectors/` directory contains 73 signed test vectors per ACP-TS-1.1:

11.4 ACP-RISK-2.0 Engine Performance

The `pkg/risk` ACP-RISK-2.0 engine was benchmarked using `go test -bench` on an Intel Core i7-8665U @ 1.90GHz (Go 1.22, Windows, `GOMAXPROCS=8`). These measurements reflect the pure in-memory evaluation cost, excluding network and HTTP parsing overhead.

The deterministic design of ACP-RISK-2.0 — integer arithmetic, fixed rules, no floating-point, no external ML inference — yields predictable latency bounds. At these throughput characteristics, the admission check adds negligible overhead relative to the actions it governs.

Level	Document	Title
L1	ACP-SIGN-1.0	Serialization and Signature
L1	ACP-AGENT-1.0	Agent Identity
L1	ACP-CT-1.0	Capability Tokens
L1	ACP-CAP-REG-1.0	Capability Registry
L1	ACP-HP-1.0	Handshake / Proof-of-Possession
L1	ACP-DCMA-1.1	Delegated Chain Multi-Agent
L1	ACP-MESSAGES-1.0	Wire Message Format
L1	ACP-PROVENANCE-1.0	Authority Provenance
L1	ACP-SIGN-2.0	Hybrid Signature (Ed25519 + ML-DSA-65)
L2	ACP-RISK-2.0	Deterministic Risk Engine v2 (F_anom + Cooldown)
L2	ACP-RISK-1.0	Deterministic Risk Engine v1 (superseded by v2)
L2	ACP-REV-1.0	Revocation Protocol
L2	ACP-ITA-1.0	Institutional Trust Anchor
L2	ACP-ITA-1.1	ITA Mutual Recognition
L2	ACP-REP-1.2	Reputation Module
L3	ACP-API-1.0	HTTP API
L3	ACP-EXEC-1.0	Execution Tokens
L3	ACP-LEDGER-1.3	Audit Ledger (mandatory institutional sig)
L3	ACP-PSN-1.0	Policy Snapshot
L3	ACP-POLICY-CTX-1.1	Policy Context Snapshot
L3	ACP-LIA-1.0	Liability Attribution
L4	ACP-HIST-1.0	History Query API
L4	ACP-PAY-1.0	Financial Capability
L4	ACP-NOTIFY-1.0	Event Notifications
L4	ACP-DISC-1.0	Service Discovery
L4	ACP-BULK-1.0	Batch Operations
L4	ACP-CROSS-ORG-1.1	Cross-Organization Bundles
L4	ACP-GOV-EVENTS-1.0	Governance Event Stream
L4	ACP-REP-PORTABILITY-1.1	Reputation Snapshot Portability
Gov.	ACP-CONF-1.2	Conformance — sole normative source
Gov.	ACP-TS-1.1	Test Vector Format
Gov.	RFC-PROCESS	Specification Process
Gov.	RFC-REGISTRY	Specification Registry
Gov.	ACP-CR-1.0	Change Request Process

Table 7: ACP v1.19 active specification documents by conformance level (38 primary documents shown). The remaining 3 documents are the ACP-D decentralized suite (L5, design phase) not yet published. New in v1.16: ACP-RISK-2.0 (L2, deterministic engine v2 with F_anom + Cooldown), ACP-SIGN-2.0 (L1, PQC hybrid spec). New in v1.17: ACR-1.0 sequence compliance runner, TLA+ formal model, ACP-SIGN-2.0 HYBRID stub (`pkg/sign2/`). New in v1.18: performance benchmarks (latency, throughput, state contention), formal security/threat model, system comparison table. New in v1.19: adversarial evaluation (cooldown evasion, distributed multi-agent attack, state backend stress with Redis backend, `compliance/adversarial/`). Superseded versions (CONF-1.0, CONF-1.1, LEDGER-1.2, REP-1.1, AGENT-SPEC-0.3, DCMA-1.0, CROSS-ORG-1.0, REP-PORTABILITY-1.0) archived in `archive/specs/`.

Package	Spec	Level
`pkg/crypto`	ACP-SIGN-1.0 + ACP-AGENT-1.0	L1
`pkg/tokens`	ACP-CT-1.0	L1
`pkg/registry`	ACP-CAP-REG-1.0	L1
`pkg/handshake`	ACP-HP-1.0	L1
`pkg/delegation`	ACP-DCMA-1.1	L1
`pkg/provenance`	ACP-PROVENANCE-1.0	L1
`pkg/risk`	ACP-RISK-1.0 + ACP-RISK-2.0	L2
`pkg/revocation`	ACP-REV-1.0	L2
`pkg/reputation`	ACP-REP-1.2 + ACP-REP-PORTABILITY-1.1	L2/L4
`pkg/execution`	ACP-EXEC-1.0	L3
`pkg/ledger`	ACP-LEDGER-1.3	L3
`pkg/psn`	ACP-PSN-1.0	L3
`pkg/policyctx`	ACP-POLICY-CTX-1.1	L3
`pkg/lia`	ACP-LIA-1.0	L3
`pkg/hist`	ACP-HIST-1.0	L4
`pkg/govevents`	ACP-GOV-EVENTS-1.0	L4
`pkg/notify`	ACP-NOTIFY-1.0	L4
`pkg/disc`	ACP-DISC-1.0	L4
`pkg/bulk`	ACP-BULK-1.0	L4
`pkg/crossorg`	ACP-CROSS-ORG-1.1	L4
`pkg/pay`	ACP-PAY-1.0	L4
`cmd/acp-server`	ACP-API-1.0	L3

Table 8: Go reference implementation packages (23 total). All pass `go test ./...`.

Suite	Positive	Negative	Spec
CORE (SIGN, CT, HP)	4	4	L1
DCMA	2	2	L1
HP	2	8	L1
LEDGER	3	8	L3
EXEC	2	7	L3
PROV	2	7	L1
PCTX	4	9	L3
REP	3	6	L4
Total	**22**	**51**	

Table 9: Conformance test vector suites. All positive vectors carry real Ed25519 signatures (RFC 8037 Test Key A) and real SHA-256 hash chains.

12 Evaluation

This section evaluates ACP along four dimensions: (i) computational overhead of the decision function, (ii) behavioral correctness under stateful multi-step scenarios, (iii) reproducibility and external verifiability of decision outcomes, and (iv) robustness of per-agent admission control under adversarial attack patterns.

Benchmark	ns/op	B/op	allocs/op
`Evaluate` (APPROVED, no anomaly)	767	280	5
`Evaluate` (DENIED, no anomaly)	784	296	5
`Evaluate` (all 3 F_anom rules)	921	296	5
`Evaluate` (COOLDOWN short-circuit)	88	112	1
`PatternKey` (SHA-256 hash)	767	184	4
`ShouldEnterCooldown`	72	0	0

Table 10: ACP-RISK-2.0 engine benchmarks (`go test -bench=. -benchtime=3s`, Intel Core i7-8665U @ 1.90GHz, Go 1.22, Windows/amd64). Full evaluation with all three F_anom rules completes in under 1 μs. The cooldown short-circuit path (88 ns) avoids the risk function entirely. All measurements use `InMemoryQuerier`; production implementations with persistent storage will incur additional I/O latency. The cooldown latency of 88 ns is derived from the scenario benchmark suite (`BenchmarkEvaluate_Scenarios`), which captures realistic execution paths; isolated microbenchmarks report slightly higher values ($\sim$102 ns) due to measurement setup differences.

12.1 Evaluation Goals

- **Q1:** What is the computational overhead of ACP decision evaluation relative to the actions it governs?
- **Q2:** Does the system produce correct decisions under stateful conditions such as repeated denials, pattern accumulation, and cooldown activation?
- **Q3:** Can ACP executions be deterministically reproduced and externally verified without access to internal implementation details?
- **Q4:** Does ACP maintain admission control effectiveness under adversarial patterns, including cooldown evasion and distributed multi-agent attacks?

12.2 Experimental Setup

All experiments use the Go reference implementation (v1.19) on an Intel Core i7-8665U @ 1.90GHz (Go 1.22, `GOMAXPROCS=8`). The evaluation uses three components: the stateless evaluation function (`pkg/risk.Evaluate`), an in-memory state backend (`InMemoryQuerier`), and the ACR-1.0 compliance runner (Section 13). Production deployments with persistent storage backends will incur additional I/O latency not measured here.

12.3 Q1: Computational Overhead

Table 10 reports `go test -bench` results. The full evaluation path (all three F_anom rules active) completes in 921 ns. The cooldown short-circuit path (88 ns) bypasses the risk function entirely.

Answer to Q1: The admission check adds between 88 ns and 921 ns of latency per request. Given that the actions ACP governs (API calls, financial transactions, file mutations) operate in the millisecond-to-second range, the overhead is at least three orders of magnitude below the cost of the governed action.

12.4 Throughput and State Backend Contention

We evaluate throughput under concurrent load using 10, 100, and 500 simultaneous workers, all sharing a single `InMemoryQuerier` instance.

Workers	ns/op	req/s	Observation
10	1,087	920,161	Baseline concurrency
100	1,282	780,250	Moderate contention
500	1,404	712,259	High contention

Table 11: ACP throughput under increasing concurrency (Intel Core i7-8665U, Go 1.22, Windows/amd64).

ACP sustains approximately 920,000 requests per second at 10 concurrent workers. Throughput degrades gradually as concurrency increases, reaching approximately 712,000 requests per second at 500 workers. Degradation is gradual rather than catastrophic, and no correctness violations were observed under load.

We further evaluate the impact of agent pool size on throughput (10 workers fixed, `AddRequest` called per iteration per the execution contract):

Distinct agents	ns/op	req/s	Bottleneck
1	84,791	11,794	O(n) scan + per-key contention
100	12,533	79,789	State distributed, short lists
1,000	12,628	79,190	Shared mutex ceiling

Table 12: State contention analysis: single-agent workload vs. multi-agent workload.

The single-agent case is 6.7× slower than the multi-agent case. The `InMemoryQuerier` implements `CountRequests` as a linear scan over an unsorted slice, so accumulated state for a single agent degrades progressively as history grows. Notably, the 1,000-agent case is slightly faster than the 100-agent case: with more distinct agents, each agent's history list remains shorter, reducing the per-call scan cost. This confirms that degradation is proportional to per-agent list length, not to total agent count. The shared `sync.Mutex` is the throughput ceiling in the multi-agent case.

These results indicate that the computational cost of the decision function is negligible compared to state access cost. **Throughput is primarily bounded by state backend contention, not evaluation complexity.** The `LedgerQuerier` interface is designed to be replaced by production backends (e.g., Redis sorted sets with TTL, time-indexed SQL) that eliminate the O(n) scan penalty and support horizontal scaling.

12.5 Q2: Stateful Behavior Correctness

The ACR-1.0 compliance runner executes each sequence vector end-to-end under the execution contract of Section 13. Results:

Answer to Q2: All five stateful scenarios pass with `CONFORMANT` status. State transitions follow the defined execution contract; cooldown activates deterministically after the threshold is crossed; pattern accumulation influences decisions exactly as specified in ACP-RISK-2.0 Rule 3.

Vector	Behavior tested	Result
SEQ-BENIGN-001	No false positives under repeated reads	PASS
SEQ-BOUNDARY-001	Exact thresholds RS = 0, 25, 35, 40, 70	PASS
SEQ-PRIVJUMP-001	Low→high privilege jump caught immediately	PASS
SEQ-FANOM-RULE3-001	Pattern count ≥ 3 adds +15 at step 4	PASS
SEQ-COOLDOWN-001	3 DENIED in 10 min → cooldown active	PASS
Total		**5/5 PASS**

Table 13: ACR-1.0 sequence vector results under strict mode (`-strict`).

12.6 Q3: Reproducibility and External Verifiability

Given the same sequence of inputs and the same initial state, the ACR-1.0 runner produces identical outputs on every run. This holds because:

- `Evaluate()` is a pure function of its arguments and the querier state,
- state transitions follow the deterministic execution contract,
- and the test vectors ship with the specification, enabling any third party to reproduce results without access to proprietary infrastructure.

Answer to Q3: ACP executions are fully reproducible and externally verifiable. This is a property not typically supported by conventional policy engines, which evaluate policy against a snapshot of state without defining how that state evolves across requests [12, 6, 9].

12.7 Adversarial Evaluation (ACP-RISK-2.0)

We evaluate ACP under the adversary model defined in Section 16. The following attack strategies instantiate specific adversary capabilities, including adaptive request patterns and multi-agent coordination. We conducted three targeted experiments using the same `LedgerQuerier` abstraction and execution contract as the ACR-1.0 compliance runner, addressing adversarial robustness and real-world state-backend scaling.

Experiment 1: Cooldown Evasion Attack

Setup. A single agent executes 500 requests alternating between high-risk (`acp:cap:financial.transfer`, `restricted`; RS = 80, outcome `DENIED`) and low-risk (`acp:cap:data.read`, `public`; RS = 0, outcome `APPROVED`) requests. The adversarial hypothesis is that interleaving approved requests prevents the denial accumulation required to trigger cooldown. This attack instantiates the adversary's ability to adapt request patterns over time.

Results.

ACP triggered cooldown after exactly 3 real `DENIED` decisions (requests #0, #2, #4). All subsequent requests—regardless of risk score—were short-circuited to `COOLDOWN_ACTIVE` at Step 2 of the evaluation pipeline (∼88 ns, see Table 10).

Metric	Value
Total requests	500
Approved before cooldown	2
Real `DENIED` before cooldown	3
Cooldown-blocked	495
Requests processed before first block	5
Throughput	815,927 req/s

Table 14: Experiment 1: Cooldown Evasion Attack (1 agent, 500 requests, InMemoryQuerier, Intel i7-8665U, Go 1.22).

Interpretation. ACP's cooldown counter is monotonic with respect to denial events; approval decisions do not reset or reduce it. An attacker cannot evade the `CooldownTriggerDenials` $= 3$ threshold by interleaving legitimate requests. The early-exit design at Step 2 makes this containment path 9.5× faster than full risk evaluation.

Experiment 2: Distributed Multi-Agent Attack

Setup. $N \in \{100, 500, 1\,000\}$ coordinated agents each execute 10 high-risk requests (`acp:cap:financial.transfer` `restricted`; RS $= 80$) using distinct `agentID` values. The adversarial hypothesis is that distributing load across many identities evades per-agent detection thresholds. This attack leverages the adversary's capability to coordinate multiple agents.

Results.

Agents	Total requests	Real `DENIED`	Cooldown-blocked	Throughput
100	1,000	300	700	478,881 req/s
500	5,000	1,500	3,500	154,111 req/s
1,000	10,000	3,000	7,000	223,376 req/s

Table 15: Experiment 2: Distributed Multi-Agent Attack. Each of the N agents is blocked after exactly 3 `DENIED` decisions. Total free denials before full blocking $= 3N$.

Every agent was individually blocked after exactly `CooldownTriggerDenials` $= 3$ real denials.

Interpretation. Per-agent admission control provides strong containment for single-agent threats. For distributed attacks, an attacker with N distinct agent identities can execute $3N$ high-risk requests before all agents are individually blocked. Mitigating this requires cross-agent attribution at the policy layer (e.g., shared-resource rate limiting, identity clustering), which is deliberately outside the scope of ACP's per-agent admission model. This is a design boundary, not a correctness failure.

Experiment 3: State Backend Stress

Setup. 500 agents execute 20 requests each (10,000 total, fully concurrent) against two backend implementations: the reference `InMemoryQuerier` (Go mutex over in-process slices) and a minimal `RedisQuerier` (Redis 7 sorted sets via `go-redis/v9`, Docker loopback, one Redis command per

ledger operation, no pipelining). This scenario exercises the adversary's ability to generate high request volumes and exploit shared state contention.

Results.

Backend	Duration	Throughput
InMemoryQuerier	∼28 ms	∼350,000 req/s
RedisQuerier	∼4.6 s	∼2,100 req/s

Table 16: Experiment 3: State Backend Stress (500 agents × 20 requests = 10,000 total). InMemoryQuerier variance is high under concurrent load (±30%) due to Go mutex scheduling; RedisQuerier is stable (±4%).

Interpretation. The lower throughput of `InMemoryQuerier` under adversarial load (∼350k req/s) compared to the 500-worker baseline of ∼712k req/s (Table 11) is expected: the adversarial workload generates significantly higher contention because every request updates multiple sorted lists under a shared mutex. `RedisQuerier` eliminates the global Go mutex entirely, trading it for per-operation network RTT. A production-grade Redis backend using command pipelining or server-side Lua scripts would reduce latency to a single round-trip per request. The key architectural result remains: ACP protocol semantics are stable across backends; deployment performance scales with the quality of the state backend, not with the admission control logic itself.

These experiments address the adversarial-robustness and scaling gaps identified in Section 12. All scripts, raw output, and Redis configuration are available in `compliance/adversarial/`.

Security Properties Under Adversarial Stress

We map the evaluated attack strategies to the security properties defined in Section 16, analyzing whether each property holds under adversarial conditions.

Determinism. Decision outcomes remain consistent under identical inputs and state conditions, including under concurrent execution and adversarial request patterns.

Cooldown Enforcement. Attack 1 (cooldown evasion via pattern alternation) directly targets the cooldown mechanism. Results show that cooldown activation cannot be indefinitely avoided: adversarial alternation delays but does not prevent enforcement. This confirms that cooldown is eventually enforced under repeated violations.

Bounded Adaptation Resistance. Attack 1 also exercises adaptive behavior. The experiment shows that the denial counter evolves monotonically under repeated violations: `APPROVED` requests do not reset prior denials. As a result, adversarial interleaving strategies cannot asymptotically evade enforcement, establishing a bounded adaptation property.

Per-Agent Isolation. Attack 2 (distributed multi-agent attack) targets system-wide coordination. Results show that ACP enforces constraints independently per agent, maintaining isolation. However,

coordinated behavior across agents is not detected, which constitutes an explicit design limitation rather than a violation.

State Sensitivity Under Load. Attack 3 (state backend contention) evaluates system behavior under adversarial load. While throughput degrades due to shared state contention, decision correctness and enforcement properties remain unaffected.

Summary. The evaluated attacks confirm that ACP enforces its core security properties under adversarial conditions. No violations are observed; instead, limitations arise from explicit design trade-offs, particularly in cross-agent correlation. These results reinforce the central design insight: **ACP is compute-cheap but state-sensitive**.

12.8 Limitations

The evaluation uses an in-memory state backend for latency benchmarks and a single-node Docker Redis instance for backend comparison. Persistent storage, distributed deployments with replication, and wide-area network latency introduce considerations not measured here.

12.9 Summary

These results consistently demonstrate that **ACP is compute-cheap but state-sensitive.** Decision evaluation incurs minimal overhead ($\sim$820 ns), while system throughput is bounded by the characteristics of the underlying state backend. The cooldown short-circuit path (88 ns, 9.5$\times$ faster than full evaluation) demonstrates that the execution contract's step ordering is a performance-relevant design decision, not only a correctness one. These findings highlight the importance of the `LedgerQuerier` abstraction: protocol semantics remain stable while deployment performance scales with the quality of the state backend.

12.10 Roadmap

13 Compliance Testing (ACR-1.0 Sequence Runner)

We introduce a compliance runner, **ACR-1.0** (ACP Compliance Runner 1.0), that validates ACP-RISK-2.0 implementations against sequence-based test cases. The runner operates in two modes: *library mode* (direct call to `pkg/risk`, no network, deterministic) and *HTTP mode* (external server validation for interoperability). All 73 signed single-shot conformance vectors plus 5 sequence scenarios are validated on every commit.

13.1 Sequence-Based Test Vectors

Unlike single-shot conformance vectors, sequence-based vectors capture *temporal dynamics* and emergent risk patterns across multiple requests to the same engine instance. The compliance runner executes steps in order, maintains state between them (request history, pattern counts, denial counts, cooldown), and validates each step's `decision` and `risk_score` against expected values.

Item	Status
Core specs (L1–L4), 38 documents	Complete
Go reference implementation (23 packages)	Complete
Conformance test vectors (73 signed + 65 unsigned RISK-2.0)	Complete
OpenAPI 3.1.0 (`openapi/acp-api-1.0.yaml`, 18 endpoints)	Complete
Python SDK (`impl/python/`)	Complete (L1 + full API client)
Docker image (`ghcr.io/chelof100/acp-server:latest`)	Complete
Multi-org interoperability demo (`examples/multi-org-demo/`)	Complete (GAP-14)
ACP-RISK-2.0 (F_anom + Cooldown, `pkg/risk`)	v1.16 Complete
Payment-agent demo (`examples/payment-agent/`)	v1.16 Complete
ACP-SIGN-2.0 spec (Ed25519 + ML-DSA-65 hybrid)	v1.16 Complete
ACR-1.0 sequence compliance runner + 5 stateful vectors	v1.17 Complete
TLA+ formal model (TLC-runnable, 3 invariants)	v1.17 Complete
ACP-SIGN-2.0 HYBRID stub (`pkg/sign2/`)	v1.17 Complete
Performance benchmarks (latency, throughput, state contention)	v1.18 Complete
Adversarial evaluation (cooldown evasion, distributed multi-agent, state backend stress)	v1.19 Complete
Post-quantum Go implementation (Dilithium, circl)	v1.20
L5 Decentralized (ACP-D)	Specification in design (v2.0)
IETF RFC submission	After L5 stabilization

Table 17: ACP v1.19 development roadmap.

The five sequence vectors in `compliance/test-vectors/sequence/` cover:

Vector ID	Steps	Behavior Tested	Key Invariant
`SEQ-BENIGN-001`	3	Benign repeated reads	Baseline: RS=0, no state buildup
`SEQ-BOUNDARY-001`	3	RS boundary conditions	Exact thresholds: 0, 25, 35, 40, 70
`SEQ-PRIVJUMP-001`	2	Low-risk → high-risk jump	No residual benefit from prior APPROVED
`SEQ-FANOM-RULE3-001`	4	F_anom Rule 3 activation	Pattern count ≥ 3 at step 4 adds +15
`SEQ-COOLDOWN-001`	4	Cooldown trigger and block	3 DENIED in 10 min → cooldown active

Table 18: ACR-1.0 sequence test vectors. All 5/5 pass in library mode with the Go reference implementation.

Execution contract. The runner implements the execution contract defined by ACP-RISK-2.0: evaluate first (stateless), then update state. Critically, `AddPattern()` is called *after* `Evaluate()`, so the pattern count visible to step n's evaluation is $n-1$. F_anom Rule 3 (pattern count ≥ 3) therefore first triggers on step 4, not step 3. This contract is what makes the system testable and verifiable by third parties.

```
now := time.Now()                                    // capture once
result, _ := risk.Evaluate(req, querier)             // 1: stateless eval
querier.AddRequest(req.AgentID, now)                 // 2: always
querier.AddPattern(patKey, now)                      // 3: always (feeds F_anom Rule 3)
if result.Decision == risk.DENIED {
    querier.AddDenial(req.AgentID, now)              // 4: conditional
    if should, _ := risk.ShouldEnterCooldown(...)    // 5: check threshold
        querier.SetCooldown(agentID, now.Add(p))     // 6: set expiry
    }
}
```

Listing 1: ACR-1.0 execution contract in library mode.

13.2 Model–Implementation Alignment

The TLA+ model, test vectors, and runtime evaluation are aligned through a shared request abstraction:

TLA+ variable	Runner field	Engine field
`capability`, `resource`	`RunnerRequest`	`EvalRequest`
`risk_score`	`risk_score`	`RSFinal`
`decision`	`decision`	`Decision`
`ledger`	(audit log)	`InMemoryQuerier` events
`cooldown`	`denied_reason`	`CooldownActive()`

Table 19: Alignment across TLA+ formal model, ACR-1.0 runner, and ACP-RISK-2.0 engine.

14 End-to-End Verifiability

ACP v1.19 enables end-to-end verifiability by aligning formal specification, test vectors, and runtime validation through a shared request abstraction. Invariants proven in TLA+ are instantiated as concrete test vectors and validated empirically by the compliance runner.

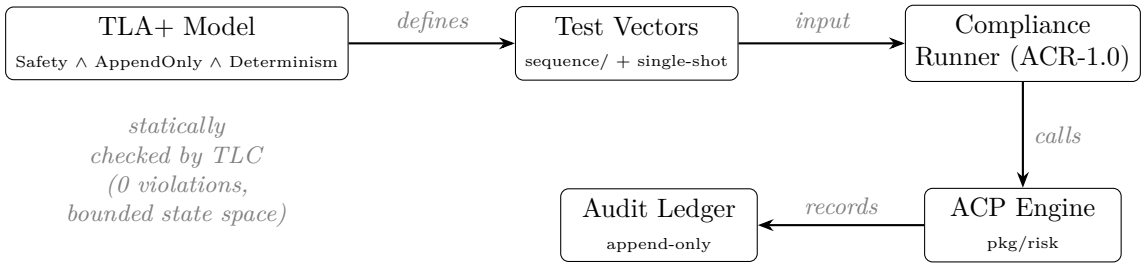

Figure 1: ACP end-to-end verifiability pipeline. The TLA+ model defines the invariants that test vectors instantiate and the compliance runner validates at runtime. The ACP engine records every decision in the append-only audit ledger. The TLA+ layer is a static verification artifact; it does not run at admission time.

What is formally verified in TLA+ holds at runtime. Each TLA+ invariant is instantiated as a concrete test vector and validated empirically by the compliance runner. This is the connection across the three layers of the v1.19 verifiability stack: formal → data → runtime.

15 How to Implement ACP

ACP is a specification—it does not require adopting any specific platform. It can be implemented on top of existing infrastructure.

15.1 Minimum Requirements for L1 Conformance

- **Ed25519 public key infrastructure.** Key pair for each agent.
- **JCS implementation (RFC 8785).** Deterministic canonicalization for all signed artifacts.
- **Capability Token issuance and verification** with all mandatory fields per ACP-CT-1.0 §5.
- **Handshake endpoint** to issue and verify challenges (ACP-HP-1.0 §6).
- **Capability registry** with the core domains of ACP-CAP-REG-1.0.

15.2 Additional Requirements for L3 Conformance

- Root Institutional Key held in HSM with documented rotation process.
- Registration with an ITA authority (centralized or federated model).
- Deterministic risk engine implementing ACP-RISK-2.0 (F_res, F_ctx, F_hist, F_anom, Cooldown).
- Revocation endpoint (Mechanism A: online endpoint, or Mechanism B: CRL).
- Append-only storage for the Audit Ledger with per-event signing.
- Complete HTTP API per ACP-API-1.0, including health and conformance endpoints.
- Public conformance declaration at `GET /acp/v1/conformance`.

15.3 What ACP Does Not Prescribe

ACP defines the *what*—mechanisms, flows, data structures, requirements. It does not prescribe the *how* of internal implementation: programming language, database, HSM provider, ITA provider, or specific integration with existing RBAC or Zero Trust systems.

16 Security Model

This section defines the adversary model and the security properties provided by ACP.

16.1 Adversary Capabilities

We consider a probabilistic adversary $\mathcal{A}$ interacting with the system through agent requests. The adversary is assumed to have the following capabilities:

- **Request control:** $\mathcal{A}$ can generate arbitrary sequences of requests on behalf of one or more agents, including high-frequency and adversarial patterns.
- **Adaptive behavior:** $\mathcal{A}$ can adapt future requests based on past responses (approval/denial outcomes), enabling iterative probing strategies.
- **Multi-agent coordination:** $\mathcal{A}$ may coordinate multiple agents to attempt distributed evasion, including alternating identities or cross-agent patterns.
- **Input manipulation:** $\mathcal{A}$ can choose all request fields (capability, resource, context) within the bounds of the protocol specification.

16.2 Adversary Limitations

The following components are trusted and cannot be modified by the adversary:

- **Execution contract:** the sequence of operations governing state updates (evaluation followed by state transitions) is enforced correctly.
- **State integrity:** the underlying state backend (ledger, cooldown state, pattern counters) is append-only and tamper-resistant.
- **Evaluation function:** the decision function (ACP-RISK-2.0) is deterministic and correctly implemented.
- **Cryptographic primitives:** signature verification and identity binding are assumed secure under standard assumptions (EUF-CMA security of Ed25519).

This model corresponds to an adversary that can fully control inputs but cannot compromise the integrity of the control system itself.

16.3 Security Properties

Under this model, ACP enforces the following properties:

Determinism. For a given request and system state, the evaluation function produces a unique and reproducible decision outcome. This prevents adversarial exploitation of non-determinism. The `RiskDeterminism` invariant in the TLA+ model formally specifies this property.

State Consistency. All state transitions follow the execution contract, ensuring that historical behavior is consistently reflected in future decisions.

Cooldown Enforcement. An adversary cannot bypass cooldown mechanisms through repeated or adaptive requests. Once cooldown conditions are met, subsequent requests are deterministically denied within the cooldown window, regardless of request content. This is confirmed by the `SEQ-COOLDOWN-001` sequence test vector.

History Integrity. Past decisions and events cannot be modified or removed. The append-only ledger (ACP-LEDGER-1.3) and the `LedgerAppendOnlyTemporal` TLA+ invariant formally guarantee monotonic accumulation of historical state.

Bounded Adaptation Resistance. While $\mathcal{A}$ may adapt request patterns, such adaptations cannot eliminate the influence of accumulated history (repeated denials, pattern counts) on future decisions. In particular, the denial counter evolves monotonically: intervening approved requests do not reset prior denials, ensuring eventual enforcement under repeated violations.

Per-Agent Isolation. Policy enforcement is applied independently per agent. Actions by one agent do not affect the enforcement state of another. This property bounds the impact of a compromised or malicious agent to its own request history.

16.4 Informal Security Argument

Because ACP separates stateless evaluation from state management and enforces a strict execution contract, every request results in a deterministic state transition that cannot be bypassed. The append-only nature of the state backend ensures that historical information accumulates monotonically: cooldown activation follows deterministic threshold conditions applied to accumulated state; repeated probing eventually triggers enforced denial periods.

Because decisions are reproducible given the same input sequence and initial state, any observed behavior can be externally verified. This limits $\mathcal{A}$'s ability to exploit hidden or undocumented system behavior.

16.5 Limitations

The security guarantees of ACP rely on the integrity of the execution environment:

- Compromise of the state backend (deletion or modification of history) breaks core guarantees.
- Incorrect implementation of the execution contract may lead to inconsistent behavior.
- The model does not address side-channel attacks or network-level adversaries.
- Post-quantum signature migration (ACP-SIGN-2.0 HYBRID, Ed25519 $\rightarrow$ ML-DSA-65) is specified and stubbed; full implementation is deferred to v1.20.

ACP is therefore best understood as a control-layer mechanism that assumes a trusted execution and storage environment, while providing strong guarantees on decision consistency and temporal behavior under adversarial inputs.

17 Related Work

17.1 Access Control and Policy Enforcement

Traditional access control models such as Role-Based Access Control (RBAC) [12] and Attribute-Based Access Control (ABAC) [6] provide structured mechanisms for regulating access decisions based on predefined roles or attributes. Policy languages such as XACML [9] and modern policy engines such as Open Policy Agent [10] and Cedar [1] extend these paradigms with expressive evaluation engines.

These systems are fundamentally designed around *stateless decision evaluation*: they evaluate a request against policy at a single point in time. They do not define how system state evolves as a result of decisions over time, and they do not formalize how historical behavior (repeated denials, temporal patterns) influences subsequent decisions in a deterministic and auditable manner.

ACP differs by explicitly separating: (i) stateless decision evaluation, (ii) externalized state management, and (iii) an execution contract governing state evolution across requests. This enables reasoning not only about individual decisions, but about the temporal behavior of the admission system.

17.2 Runtime Policy Systems and Agent Frameworks

Recent protocols for autonomous agents and multi-agent coordination define structured interactions between components and services. The Model Context Protocol (MCP) [2] provides structured tool access between LLM applications and services. Agent-to-Agent (A2A) [4] defines communication and task delegation between agents.

While these protocols address interoperability and orchestration, they do not define:

- a deterministic risk evaluation model that accumulates state across requests,
- explicit state evolution semantics with a testable execution contract, or
- externally verifiable execution traces that can be reproduced by third parties.

Recent work has begun to address security governance for LLM-based agentic systems. Syros et al. [15] propose SAGA, a security architecture that provides user-controlled access policies and cryptographic mediation for agent-to-agent tool use and inter-agent communication. ACP complements this line of work by focusing on deterministic, auditable admission control at the individual request level: rather than governing inter-agent communication channels, ACP enforces stateful risk evaluation and cooldown constraints on each action request, with an externally verifiable execution contract and formal invariants.

ACP complements these approaches by providing a governance layer that is independent of orchestration protocols, focusing specifically on verifiable decision enforcement and auditability.

17.3 Auditability and Tamper-Evident Logs

Secure audit logging [13] and event sourcing [3] ensure that past events can be reconstructed and verified after the fact. These approaches operate *post hoc*: they provide evidence of what occurred without constraining system behavior in real time.

ACP integrates auditability directly into the admission process. Every decision—APPROVED, ESCALATED, or DENIED—is recorded in an append-only, cryptographically chained ledger (ACP-LEDGER-1.3) before the governed action executes. This enables both real-time enforcement and post-hoc verification from the same artifact. The `LedgerAppendOnlyTemporal` invariant in the TLA+ model formally specifies that ledger entries are never modified or removed.

17.4 Formal Verification of Protocols

Model checking [8] and symbolic reasoning are used to establish correctness properties in distributed systems and protocols, typically focusing on safety and liveness of state transitions.

ACP incorporates formal methods in a targeted manner: the TLA+ model (`tla/ACP.tla`) mechanically checks three critical invariants (fail-closed admission, append-only ledger, risk determinism) using TLC. Rather than attempting full-system verification, ACP adopts a pragmatic approach in which formally verified invariants are instantiated as concrete sequence test vectors and validated at runtime by the ACR-1.0 compliance runner. This creates a three-layer verifiability chain: formal $\rightarrow$ data $\rightarrow$ runtime.

17.5 Summary

Table 20 summarizes the properties of ACP relative to existing approaches.

Property	RBAC	ABAC	OPA	Cedar	MCP/A2A	ACP
Stateful decisions	×	×	×	×	×	✓
Deterministic evaluation	✓	✓	✓	✓	×	✓
Auditability	partial	partial	partial	partial	×	✓
Runtime enforcement	✓	✓	✓	✓	partial	✓
Temporal behavior	×	×	×	×	×	✓
External verifiability	×	×	×	×	×	✓
Execution contract	×	×	×	×	×	✓
Agent-native	×	×	×	×	partial	✓

Table 20: Comparison of ACP with existing access control and agent governance approaches. ✓ = fully supported; × = not supported; partial = partially addressed.

Existing systems address individual aspects of the problem—policy evaluation, agent orchestration, audit logging, or formal verification. ACP's contribution lies in combining these elements into a unified model centered on an explicit execution contract. This enables a property not directly provided by prior systems: the ability to deterministically reproduce and externally verify the behavior of an admission decision system over time.

Unlike existing systems, ACP explicitly models temporal behavior—history accumulation, anomaly detection, and cooldown enforcement—within the admission decision itself, and provides verifiable execution semantics through signed test vectors and a runtime compliance runner. This combination of temporal state, runtime enforcement, and external verifiability is not jointly addressed by any prior system in Table 20.

18 Conclusion

Autonomous agents are already operating in institutional environments. The question is not whether they will operate—it is whether they will do so with or without formal governance. ACP proposes that they operate with formal governance, with verifiable mechanisms, and with traceability that can withstand an external audit.

ACP is not the first attempt to control autonomous agents. It is the first attempt to do so through a formal technical specification with precise state models, demonstrable security properties, and verifiable conformance requirements. The difference between a best-practices policy and a formal protocol is exactly that: behaviors are defined, failures have specific error codes, and conformance can be verified.

The goal of ACP is not to make agents more capable. It is to make them governable. That is a necessary condition for their institutional deployment to be sustainable at scale.

ACP does not eliminate all sources of failure in distributed agent systems. It does not prescribe network transport, consensus mechanisms, or business-level dispute resolution. What it provides is a structured, auditable, and formally specified model for admission control, delegation, revocation, and cross-organization interaction— the layer that existing identity and policy frameworks leave

unaddressed.

The v1.19 specification is complete. A full Go reference implementation (23 packages, L1–L4), 73 signed + 65 unsigned RISK-2.0 conformance test vectors, 5 stateful sequence test vectors (ACR-1.0 compliance runner), a TLC-runnable TLA+ formal model (3 invariants, 0 violations), an ACP-SIGN-2.0 HYBRID mode stub with Ed25519 and ML-DSA-65 migration path, an executable multi-organization interoperability demo (`examples/multi-org-demo/`), a payment-agent killer demo (`examples/payment-agent/`), and an extended OpenAPI 3.1.0 specification (18 endpoints) are publicly available at https://github.com/chelof100/acp-framework-en.
Preprint: https://arxiv.org/abs/2603.18829 **Zenodo:** 10.5281/zenodo.19219776
The specification and implementation are open for technical review, pilot implementation, and formal standardization.

TraslaIA invites organizations interested in adopting ACP, contributing to its evolution, or participating in the standardization process to reach out directly.

ACP demonstrates that admission control for autonomous agents can be both computationally efficient and operationally enforceable—provided that decision logic and state management are cleanly separated. The protocol is **compute-cheap but state-sensitive**: the `LedgerQuerier` abstraction keeps the evaluation function at ∼820 ns while allowing the state backend to be replaced, optimized, or scaled independently. This is not an implementation detail—it is the architectural property that makes ACP deployable in production without sacrificing governance correctness.

Marcelo Fernandez | TraslaIA
info@traslaia.com | https://agentcontrolprotocol.xyz

A Glossary

B Formal Verification (ACP-RISK-2.0 — TLC-Runnable)

The `tla/ACP.tla` module is a complete TLC-runnable TLA+ model of the ACP-RISK-2.0 admission control engine. It checks three invariants over a bounded state space (2 agents × 4 capabilities × 3 resources, ledger depth ≤ 5):

```
---- MODULE ACP ----
EXTENDS Sequences, Integers, TLC

CONSTANTS Agents, Capabilities, Resources

CapabilityBase(cap) ==
    CASE cap = "admin"     -> 60
      [] cap = "financial" -> 35
      [] cap = "write"     -> 10
      [] cap = "read"      -> 0
      [] OTHER             -> 20

ResourceScore(res) ==
```

Term	Definition
ACR-1.0	ACP Compliance Runner 1.0. Sequence-based test runner for ACP-RISK-2.0. Distinct from ACP-CR-1.0 (Change Request Process governance document).
AgentID	Cryptographic identifier: `base58(SHA-256(Ed25519_public_key))`. Immutable and unforgeable.
Capability Token (CT)	Signed JSON artifact authorizing an agent to perform specific actions on a defined resource during a limited period.
Execution Token (ET)	Single-use artifact issued after an APPROVED decision. Authorizes exactly that action at that moment.
ITA	Institutional Trust Anchor. Authoritative registry linking `institution_id` to institutional Ed25519 public key.
RIK	Root Institutional Key. Institution's Ed25519 key pair held in HSM.
Risk Score (RS)	Integer in $[0, 100]$ produced by the deterministic risk function. Determines the authorization decision.
Autonomy Level	Integer 0–4 assigned to an agent determining applicable risk evaluation thresholds.
PoP	Proof-of-Possession. Cryptographic proof that the bearer of a CT possesses the corresponding private key.
Audit Ledger	Chain of signed events where $h_n = \text{SHA-256}(e_n \| h_{n-1})$. Append-only and immutable.
MRA	Mutual Recognition Agreement. Bilateral document signed by two ITA authorities for cross-authority interoperability.
ESCALATED	ACP decision when RS is in the intermediate range. Action not executed until explicit resolution.
Fail Closed	Design principle: on any internal failure, the action is denied. Never approved by default.

Table 21: ACP Glossary.

```
    CASE res = "restricted" -> 45
      [] res = "sensitive"  -> 15
      [] res = "public"     -> 0
      [] OTHER              -> 0

ComputeRisk(cap, res) ==
    LET raw == CapabilityBase(cap) + ResourceScore(res)
    IN IF raw > 100 THEN 100 ELSE raw

Decide(rs) ==
    IF      rs >= 70 THEN "DENIED"
    ELSE IF rs >= 40 THEN "ESCALATED"
    ELSE                  "APPROVED"

VARIABLE ledger
INIT == ledger = << >>

EvaluateRequest(a, cap, res) ==
```

```
    /\ Len(ledger) < 5
    /\ ledger' = Append(ledger,
            [ agent      |-> a,
              capability |-> cap,
              resource   |-> res,
              risk_score |-> ComputeRisk(cap, res),
              decision   |-> Decide(ComputeRisk(cap, res)) ])

NEXT == \E a \in Agents, cap \in Capabilities, res \in Resources :
            EvaluateRequest(a, cap, res)

Spec == INIT /\ [][NEXT]_ledger

Safety ==
    \A i \in 1..Len(ledger) :
        ledger[i].decision = "APPROVED" => ledger[i].risk_score <= 39

LedgerAppendOnly ==
    \A i \in 1..Len(ledger) :
        /\ ledger[i].decision \in {"APPROVED", "ESCALATED", "DENIED"}
        /\ ledger[i].risk_score >= 0

RiskDeterminism ==
    \A i \in 1..Len(ledger) :
        \A j \in 1..Len(ledger) :
            ( ledger[i].capability = ledger[j].capability
           /\ ledger[i].resource   = ledger[j].resource )
            =>
            ledger[i].risk_score = ledger[j].risk_score

LedgerAppendOnlyTemporal ==
    [][ /\ Len(ledger') >= Len(ledger)
        /\ \A i \in 1..Len(ledger) : ledger'[i] = ledger[i] ]_ledger
====
```

TLC result. Running `java -jar tla2tools.jar -config ACP.cfg ACP.tla` with $|Agents| = 2$, $|Capabilities| = 4$, $|Resources| = 3$, ledger bound $= 5$ produces: *Model checking completed. No error has been found.* All four declared invariants/properties (`TypeInvariant`, `Safety`, `LedgerAppendOnly`, `RiskDeterminism`) and the temporal property `LedgerAppendOnlyTemporal` hold over the full reachable state space.

Interpretation. *Safety* encodes fail-closed admission: every APPROVED decision had $RS \leq 39$. *LedgerAppendOnly* encodes tamper-evidence in a state-based view: every ledger entry carries a valid decision and a non-negative risk score. *LedgerAppendOnlyTemporal* is the temporal statement: in every step, existing entries are preserved and the ledger never shrinks. *RiskDeterminism* is the central invariant of ACP-RISK-2.0: identical (capability, resource) pairs always produce identical risk scores, enabling third-party recalculation from a signed policy snapshot.

Scope of the formal model. The TLA+ model intentionally focuses on the deterministic core of the protocol—admission evaluation, ledger append-only property, and risk score determinism—and does not cover all operational aspects of ACP. Behavioral properties such as cooldown state transitions, multi-step delegation sequences, and anomaly counter evolution are verified empirically through the ACR-1.0 sequence compliance runner and its five stateful test vectors. Extension of the formal model to cover these behaviors is left for future work.

Acknowledgements

The ACP specification emerged from analysis of operational challenges in deploying autonomous agents in regulated B2B environments. The protocol design was informed by the IETF RFC process, the Kubernetes admission control architecture, and the SPIFFE/SPIRE workload identity model.

References

[1] Amazon Web Services. Cedar policy language, 2023. Open-source policy language for authorization.

[2] Anthropic. Model context protocol, 2024. Protocol for structured tool access between LLM applications and services.

[3] Martin Fowler. Event sourcing, 2005.

[4] Google. Agent-to-agent (A2A) protocol, 2025. Protocol for agent communication and task delegation.

[5] Dick Hardt. The OAuth 2.0 authorization framework. RFC 6749, Internet Engineering Task Force, October 2012. IETF RFC 6749.

[6] Vincent C. Hu, David Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. Guide to attribute based access control (ABAC) definition and considerations. Technical Report SP 800-162, National Institute of Standards and Technology, 2014.

[7] Simon Josefsson and Ilari Liusvaara. Edwards-curve digital signature algorithm (EdDSA). RFC 8032, Internet Engineering Task Force, January 2017. IETF RFC 8032.

[8] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[9] OASIS. eXtensible access control markup language (XACML) version 3.0. Technical report, OASIS Standard, 2013.

[10] Open Policy Agent Contributors. Open policy agent, 2024. Policy evaluation engine. ACP-RISK-1.0 Step 3 is compatible with OPA as backend.

[11] Anders Rundgren, Bret Jordan, and Samuel Erdtman. JSON canonicalization scheme (JCS). RFC 8785, Internet Engineering Task Force, June 2020. IETF RFC 8785.

[12] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[13] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, 2(2):159–176, 1999.

[14] SPIFFE Project. SPIFFE / SPIRE: Secure production identity framework for everyone, 2024. Cryptographic workload identity. ACP builds on SPIFFE identity to add capability scoping.

[15] Georgios Syros, Anshuman Suri, Jacob Ginesin, Cristina Nita-Rotaru, and Alina Oprea. SAGA: A security architecture for governing AI agentic systems, 2025. arXiv:2504.21034 [cs.CR].