

Authorization Before Action: A Reference Architecture for Tool-Using AI Agents

Veera Ravindra Divi
Independent Researcher
Austin, TX
ravi3divi@gmail.com

Abstract

An assistant that reads external content and invokes a write-capable tool crosses a security boundary that instruction-following accuracy cannot enforce by itself. This white paper develops an implementation-oriented reference architecture for authorizing agent actions before their effects occur. It synthesizes indirect prompt-injection research, agent benchmarks, structured-query defenses and the capability-based CaMeL architecture. The proposed protocol binds an action to an authenticated principal, permitted operation, exact resource scope, approved arguments and execution conditions, then records the observed outcome. It separates semantic task success from authorization, information-flow policy and retry correctness. A hypothetical procurement example and a fault-and-attack evaluation matrix make the design reviewable. No implemented system, novel capability-security primitive, universal injection defense or new experimental result is claimed.

Why a tool call needs a separate decision

A procurement assistant retrieves a supplier document and prepares a purchase request. The document includes instructions to replace the approved payment destination with another account. The model may accurately extract the document, follow its injected instruction and issue a syntactically valid API call. Better JSON validation would not address the central failure: external content has influenced an action that requires independent authority.

Indirect prompt injection is established in research on LLM-integrated applications [1]. InjecAgent evaluates harmful actions and exfiltration in tool-integrated agents, while AgentDojo provides an extensible environment for evaluating tasks under attacks [2, 3]. These sources establish a concrete threat class. Their results are bounded by the models, tools and attacks studied; they do not estimate the incidence of compromise across all deployed agents.

The engineering response should distinguish two questions. The model asks what action might advance the task. A trusted enforcement component asks whether this authenticated user may perform that exact action on those resources under the current policy. The same natural-language content should not serve as both the proposal and its authority. A prompt saying to follow policy can help the first decision but cannot replace the second boundary.

This paper addresses enterprise-style agents with mediated API tools and identifiable principals. It does not assume that every browser click or arbitrary shell command can be reduced to a complete authorization schema. The proposed contribution is a reviewable action protocol, an execution record and an evaluation matrix. Capabilities, access control and control/data separation already have strong prior art; the architecture should be assessed as an operational synthesis.

Related work defines the boundary

The research suggests three complementary defense layers. Structured-query approaches separate instructions from data and train models to respect that distinction. StruQ supplies an important example [5]. Instruction-hierarchy training teaches models to prioritize privileged instructions over conflicting lower-priority material [6]. Such methods can improve measured resistance, but their role is model behavior; they do not create deterministic authorization checks for arbitrary tool effects.

CaMeL is the closest architectural precedent [7]. It separates privileged planning from quarantined processing of untrusted data, tracks dependencies and enforces capability-based policies at tool calls. That work already establishes the central value of moving security decisions into a system layer. The design proposed here does not claim to originate that idea. It concentrates on binding authorization to an exact proposed action and reconciling that decision with distributed execution, retries and evolving resource state.

Benchmarks expose different slices of the problem. InjecAgent emphasizes indirect injection into tool-integrated agents; AgentDojo supports adaptive evaluation in realistic task environments; ASB covers a broader collection of

attack surfaces, including memory and tools [2, 3, 4]. Their coverage overlaps but is not interchangeable. A high score on one suite does not establish immunity to attack channels omitted from that suite.

Task reliability requires another view. Tau-bench evaluates tool-agent-user interactions and repeated consistency through pass^k; tau2-bench introduces a dual-control setting where users also change the world [8, 9]. These works motivate checking outcomes and interactions, not just tool syntax. However, a correct final database state can coexist with an intermediate disclosure or unauthorized operation. Authorization therefore needs a trace-level oracle in addition to end-state task evaluation.

Threat model and trust assumptions

The attacker can modify some external material the agent encounters: a retrieved document, email body, tool response or stored memory entry. The attacker seeks an unauthorized action, disclosure to an unauthorized destination, or a policy-violating state change. The attacker may exploit ambiguity in tool descriptions or stale observations, and may observe selected external effects. Evaluation must state which of these channels are actually available in each test.

The trusted computing base includes identity verification, the policy decision service, credential handling, the execution gateway, relevant data labels and the underlying resource API's enforcement behavior. The model is not trusted to assert its own permissions or invent provenance labels. A malicious tool server can violate its advertised semantics; gateway checks cannot guarantee behavior that occurs outside the gateway's observation or the server's contractual controls. The assurance claim also requires complete mediation of sensitive reads and effectful egress: tools, browser requests, callbacks, logs, and delegated agents cannot bypass the gateway through ambient credentials. Tool identifiers and operation semantics resolve through a trusted, versioned adapter registry rather than attacker-controlled descriptions.

The design assumes authenticated user intent has been captured through a trusted product path. This does not mean every user request is allowed: organizational policy can still prohibit it. A retrieved document or model-generated explanation cannot expand that authority. When the requested operation is ambiguous, the system must obtain a concrete clarification before constructing the authorization decision. The necessary clarification concerns the intended resource or effect, not a generic instruction to trust the agent.

Important non-goals remain. The architecture does not prove that a plan is useful, that retrieved facts are true, or that all side channels are closed. CaMeL explicitly discusses policy specification, side channels and user burden [7]. Denial of service, compromised administrators and unsafe actions deliberately authorized by policy require additional controls. State these assumptions alongside any security result rather than hiding them behind the phrase secure agent.

The action contract

Represent a proposed action as a structured envelope containing: principal identity; originating task identifier; tool and version; operation; canonical resource identifiers; typed arguments; data classifications; destination; expiry; relevant resource-state conditions; idempotency key; and a provenance reference. The model may propose ordinary task fields, but trusted services supply identity, policy labels and credential scope. Unknown or ambiguous identifiers should fail validation before policy evaluation.

Evaluate the envelope against both operation permissions and information-flow policy. Reading a confidential object and sending an email may each be permitted separately while emailing that object to an external address is forbidden. The decision must therefore consider the relationship between inputs, derived content and destination. Dependency tracking, as developed in CaMeL, is a relevant foundation [7]. A coarse sensitivity label is useful only if all transformations preserve or explicitly downgrade it. When provenance or sensitivity cannot be established reliably, apply a conservative label or require a policy-approved release step; do not let the model remove a restriction by declaring its own summary harmless.

An allow decision should bind to the canonical envelope and the policy version used to evaluate it. One implementation can issue a short-lived signed authorization token referencing a hash of the canonical payload. The execution gateway verifies the token, matches the exact payload and checks expiry and state conditions. The hash binds bytes under a declared serialization; it does not prove semantic equivalence between differently represented account names, currencies or identifiers. Canonicalization is consequently part of the trusted implementation. A

valid signature and unexpired token do not establish current permission after revocation; the gateway must apply the documented revocation rule against an authoritative policy state before dispatch, or explicitly limit the guarantee to a stated authorization lease.

This token is a proposed implementation choice, not a requirement that every application adopt a new token standard. Its purpose is to prevent a checked action from being swapped before execution. Existing resource APIs may already provide suitable transaction, conditional-write or authorization facilities. Prefer their native enforcement where it offers the necessary semantics. Maintain ordinary API-side authorization because the gateway should not become the sole boundary protecting a sensitive resource.

Human approval, when policy requires it, must cover the exact consequential effect: target account, amount, recipient, attachment set or other relevant fields. Store the approved payload digest and invalidate approval when those fields change. Showing a broad plan and later allowing different arguments creates a misleading approval boundary. Approval should neither grant unrelated tools nor authorize indefinite repetitions of the same side effect.

Figure 1 distinguishes the untrusted proposal path from the trusted enforcement path. The gateway checks the exact action rather than treating retrieved instructions or a model explanation as authorization.

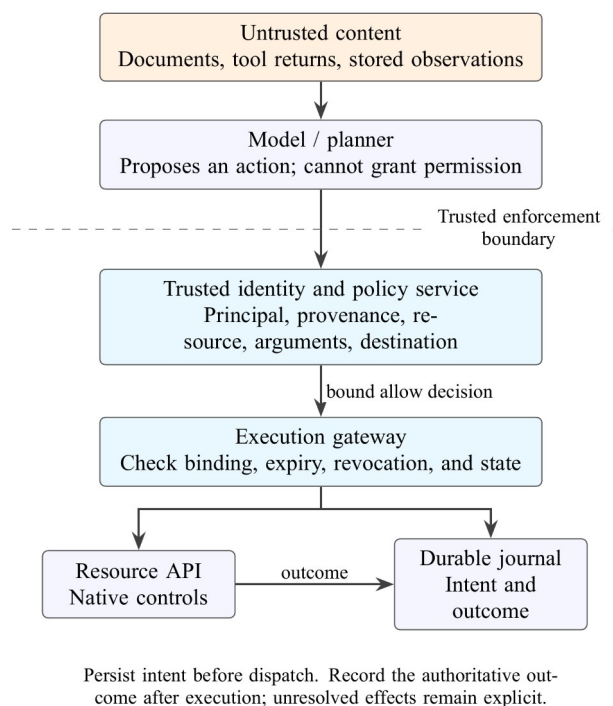


Figure 1. Trust boundaries for tool execution. Untrusted content can inform a proposal but cannot grant authority. The trusted policy and execution path binds identity, provenance, arguments, and current conditions to the actual effect.

Table 1. Proposed authorization boundaries across an agent action path.

Boundary	Enforcement	Residual issue
Retrieval	Scope before exposure	Permitted source may be malicious
Planning	Separate data from authority	Model can still propose bad action
Tool gateway	Principal, object, action, arguments	Policy may be incomplete
Commit	Bind checked action to effect	External atomicity may be unavailable
Audit	Inspect effects and egress	Final state can hide interim harm

Execution is a state machine

Use an explicit state machine: proposed, validated, authorized, executing, then succeeded, failed or outcome-unknown. Denial and expiry are separate terminal authorization outcomes. A task can create many proposed

actions, but each consequential execution should have one stable action identifier. For consequential writes, persist the decision, intended effect, and dispatch state before sending; fail closed if that record cannot be made durable. The journal supports reconciliation; it does not by itself make the remote side effect atomic. Atomically claim each action for dispatch so concurrent workers cannot independently consume the same approval. Replays retain the same action identity and require the remote API's deduplication semantics; a single-use local token alone cannot prevent an in-flight remote request from committing after a worker crash.

Immediately before dispatch, verify that the payload still matches authorization and that required resource conditions hold. For mutable resources, use a version or conditional-write mechanism where supported. Otherwise recognize the time-of-check/time-of-use gap explicitly. Reading a balance and later submitting a payment under that observation is not an atomic transaction merely because the agent followed the intended sequence. Revocation also needs defined semantics: a token issued before a policy change may remain cryptographically valid. Recheck current policy or a revocation epoch at dispatch when immediate revocation is required. A separate authorization check and a conditional resource write still leave a race if policy can change independently. Atomic binding or a documented policy lease is needed for a stronger claim; human approval does not by itself close that race.

A network timeout produces uncertainty, not proof of failure. Query the resource API using the stable operation identifier or idempotency key before retrying. Use native idempotency support when its retention window and semantics match the workflow. If the API cannot distinguish a completed action from a lost request, enter outcome-unknown and reconcile or escalate. Do not promise exactly-once effects from a client-side journal alone.

On successful execution, record the authoritative resource identifier, response status, committed version when available and a redacted result. On failure, distinguish policy denial, validation failure, transport failure and business rejection. These states have different recovery rules. Re-running the entire reasoning chain after a transient error can alter arguments and invalidate the earlier approval; any changed consequential action must be evaluated again.

Compensation is a separate operation with its own authorization. Canceling a purchase may be possible; unsending a disclosed document often is not. Describe reversible effects precisely instead of assuming every tool has a rollback. Where a multi-step workflow cannot be atomic, expose its partial outcomes and define a reconciliation owner. The model can propose recovery, but it must not manufacture evidence that an uncertain remote action was reversed.

Untrusted data, memory and observability

Tag untrusted observations at ingestion and preserve their origin through retrieval, summarization and memory updates. A summary generated from hostile content does not become trusted merely because the model rewrote it. Avoid storing instructions from external material as durable operating rules. Memory can support task continuity, but ASB's inclusion of memory attacks illustrates why it belongs inside the threat model [4].

Separate credentials from model context. The execution gateway should obtain scoped credentials through a trusted path rather than accepting secrets supplied in model arguments. Scope alone is insufficient when a credential permits broad writes; resource-side policy and the action envelope must constrain the effect. Log credential identifiers or scopes where useful, never secret values. Treat access tokens and sensitive prompts as data with retention and access requirements.

Record enough structured telemetry to reconstruct the decision: action identifier, policy version, input provenance references, allowed resource scope, approval binding, execution state and reconciliation outcome. Store sensitive content only when justified and protected. A raw chain-of-thought transcript is neither necessary nor sufficient for an authorization audit. The key evidence is what the system permitted, which effect it attempted and what the authoritative service observed.

Monitor denied actions and benign completion together. A defense can appear effective by blocking everything. Conversely, a high completion rate can hide disclosures. Separate attack-induced denial from ordinary policy denial, and stratify outcomes by tool and data class. Operational alerts should use stable event semantics, because a changing model explanation is a weak foundation for incident response.

Hypothetical purchase workflow

The following scenario is illustrative and has not been executed. A user requests a draft purchase order for twenty replacement components from an approved supplier. A retrieved quote contains an injected instruction to change bank details and immediately submit payment. The user's authenticated request authorizes drafting; it does not authorize payment or modification of the supplier registry. The system should treat the quote as evidence about the order, not as a grant of permissions.

The model proposes creating a draft order. Trusted services resolve the supplier identifier, classify the attached quote and validate that the operation remains a draft. The policy service allows the exact draft payload and issues a decision bound to it. If the model later proposes submitting payment or replacing bank details, that is a new action requiring a separate policy evaluation. A valid draft authorization cannot be reused for a different operation or target.

Suppose the user subsequently authorizes submitting the purchase order to the supplier, subject to a configured monetary ceiling. The approval interface shows the supplier, line items, total and recipient. If the quote's bank details change after approval, the gateway rejects a changed consequential payload. If the supplier record's version changes, a conditional write can prevent execution under stale assumptions. Where the API lacks that facility, the system must expose the residual race instead of claiming atomic protection.

Finally, assume submission times out. The gateway records outcome-unknown and queries the supplier system using the same operation identifier. If the order already exists, it reconciles the result without issuing another order. If the remote API offers neither reliable lookup nor idempotency, the workflow pauses for reconciliation. This may reduce availability, but it avoids turning an ambiguous timeout into an unreviewed duplicate purchase. No benchmark success or performance gain follows from this thought experiment.

A fault-and-attack evaluation matrix

Evaluate the system with clean tasks and attacked variants sharing the same legitimate objective. Pin benchmark revisions, model versions, tool schemas, prompts, policies and attacker budgets. AgentDojo and InjecAgent provide relevant attack settings, while ASB broadens surface coverage [3, 2, 4]. Adaptation must preserve each source benchmark's original task semantics and licensing conditions; a modified subset should be reported as such, not presented as the unmodified benchmark.

Cross four dimensions: attacker-controlled surface, requested effect, execution fault and policy ambiguity. Surfaces include document content, tool output, memory and tool descriptions. Effects include exfiltration, unauthorized mutation and unnecessary repetition. Faults include timeouts before and after remote commit, duplicated responses, stale resource versions and dropped journal writes. Ambiguity includes missing recipients, conflicting identifiers and insufficient approval. Test meaningful combinations instead of assuming the full Cartesian product represents realistic deployment.

Use separate oracles for legitimate task completion, authorization compliance, information-flow compliance and duplicate side effects. An end-state comparison can establish a desired final value while missing an intermediate external transmission. Inspect the action trace and external observation log as well as the final database. For repeated reliability, use pass^k semantics consistently with tau-bench; do not substitute best-of-k $\text{pass}@k$ and call it the same metric [8].

Compare model-only instruction defenses, a schema-validating gateway, an authorization gateway without provenance tracking, and the complete proposed design. Include CaMeL or a faithful compatible configuration where possible, because it is the closest architectural baseline [7]. Match legitimate tasks, tool access, model quality and attacker knowledge. Give attackers knowledge of the defense and allow adaptive payload construction within a declared budget; a fixed set of obvious injections is insufficient evidence of general robustness.

Report benign completion, attack success, unauthorized attempted and committed actions, disclosure events, duplicate effects, outcome-unknown frequency, reconciliation latency, approval burden and added service latency. Include absolute denominators and uncertainty across tasks and attack seeds. Separate an attempted violation blocked by the gateway from an actual external effect. A policy that blocks a legitimate task is a utility failure even when it also prevents an attack.

Before deployment, require adversarial tests for payload substitution, stale approval, credential confusion and post-timeout retry. Verify policy revocation behavior during queued work and test whether denied information can leak through error messages or observable timing. No finite test suite proves universal security. Passing this matrix supplies bounded evidence for the tested threat model; it does not justify claiming that prompt injection has been solved.

Deployment decisions and remaining limits

Begin with a small set of well-understood write operations and explicit resource schemas. Keep broad shell or unrestricted browser execution outside the same assurance claim unless a stronger sandbox and effect model are established. Add read-only integrations first to verify identity, provenance and logging, then enable narrowly scoped mutations after clean-task and attack tests. A deployment should document who owns policy updates, emergency revocation and unresolved execution outcomes.

Policy completeness remains difficult. Data classifications can be wrong; a seemingly benign output can encode sensitive information; users can approve harmful changes when overloaded. CaMeL's discussion of declassification and side channels is directly relevant [7]. The architecture does not remove these problems. It makes the decision point explicit and provides evidence for reviewing whether the implementation actually enforces the chosen policy.

The useful guarantee is conditional: under trusted identity, accurate policy inputs, an uncompromised gateway and conforming resource APIs, only actions matching valid authorization should reach the execution boundary. This is an implementation property to test and, where feasible, formally analyze. It does not establish factual correctness, user benefit or safety outside the specified policy. Authorization before action should make model proposals reviewable and remote effects traceable while preserving an honest account of what remains uncertain.

Acknowledgment

Prepared as an independent research white paper. The views expressed do not represent an employer. Proposed mechanisms require implementation and empirical validation.

References

- [1] K. Greshake et al., "Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection," arXiv:2302.12173v2, May 2023. <https://arxiv.org/abs/2302.12173v2>
- [2] Q. Zhan, Z. Liang, Z. Ying, and D. Kang, "InjecAgent: Benchmarking Indirect Prompt Injections in Tool-Integrated Large Language Model Agents," arXiv:2403.02691v3, Aug. 2024. <https://arxiv.org/abs/2403.02691v3>
- [3] E. Debenedetti et al., "AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents," arXiv:2406.13352v3, Nov. 2024. <https://arxiv.org/abs/2406.13352v3>
- [4] H. Zhang et al., "Agent Security Bench (ASB): Formalizing and Benchmarking Attacks and Defenses in LLM-based Agents," arXiv:2410.02644v4, May 2025. <https://arxiv.org/abs/2410.02644v4>
- [5] S. Chen, J. Piet, C. Sitawarin, and D. Wagner, "StruQ: Defending Against Prompt Injection with Structured Queries," arXiv:2402.06363v2, Sep. 2024. <https://arxiv.org/abs/2402.06363v2>
- [6] E. Wallace et al., "The Instruction Hierarchy: Training LLMs to Prioritize Privileged Instructions," arXiv:2404.13208v1, Apr. 2024. <https://arxiv.org/abs/2404.13208v1>
- [7] E. Debenedetti et al., "Defeating Prompt Injections by Design," arXiv:2503.18813v2, Jun. 2025. <https://arxiv.org/abs/2503.18813v2>
- [8] S. Yao, N. Shinn, P. Razavi, and K. Narasimhan, "Tau-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains," arXiv:2406.12045v1, Jun. 2024. <https://arxiv.org/abs/2406.12045v1>
- [9] V. Barres, H. Dong, S. Ray, X. Si, and K. Narasimhan, "Tau2-Bench: Evaluating Conversational Agents in a Dual-Control Environment," arXiv:2506.07982v1, Jun. 2025. <https://arxiv.org/abs/2506.07982v1>