International Journal of AI, Big Data, Computational and Management Studies
Noble Scholar Research Group | Volume 7, Issue 4, PP 4-15, 2026
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V7I4P102

Original Article

Separating Evidence from Executability: A GraphRAG and Deterministic Validation Architecture for Autonomous Agents

Palani Ravindran
Independent Researcher, USA.

Received On: 24/08/2026 ***Revised On: 22/09/2026*** ***Accepted On: 29/09/2026*** ***Published On: 04/10/2026***

Abstract: *Autonomous LLM agents are increasingly issued tool calls that mutate real systems provisioning infrastructure, moving funds, updating clinical records yet most agent architectures conflate the retrieval that informs a decision with the executability of the action that follows, and the resulting probabilistic substrate leaks even when the model's own reasoning correctly identified the risk. This paper proposes an architectural separation. An evidence plane, implemented as a GraphRAG store, supplies the planner with typed entities and relationships in the ordinary way; a deterministic validation plane re-queries the same graph and decides, per proposed tool call, whether it is executable. Probabilistic reasoning proposes an action, while deterministic software decides whether that action is executable. We formalize the interface as an executability predicate composed of schema, permission, precondition, and graph-fact consistency subpredicates, and we specify a mapping from check outcomes to four action paths accept, reject, correct, and escalate. The contribution is a design-science artifact: a reference architecture, a formal interface, and a JSON/YAML tool-and-rule specification deployable behind an MCP-compatible registry, illustrated through worked synthetic examples covering each of the four action paths.*

Keywords: *GraphRAG, Autonomous Agents, Deterministic Validation, Evidence-Based Reasoning, Agentic AI, Knowledge Graphs, Retrieval-Augmented Generation, Execution Validation, AI Reliability, and Decision Verification.*

1. Introduction

Autonomous LLM agents are being shipped into workflows where a single tool call can move money, mutate patient records, provision cloud resources, or dispatch a physical actuator. The prevailing safety story around these agents has been a story about the model: better instruction-following, sharper refusals, sturdier system prompts, and output classifiers stationed at the mouth of the pipeline. That story is incomplete. The failure mode that keeps recurring in incident reports is not that the model failed to *recognize* a bad action it is that the agent recognized the risk, deliberated correctly in natural language, and then still emitted an unsafe tool call anyway. Recent evaluations of reasoning-based guardrails formalize this phenomenon as a *deliberation-to-enforcement gap*: safety reasoning and enforcement live in the same probabilistic substrate, and the substrate leaks [1]. Prompt-layer filters, tool allowlists, and data-loss-prevention hooks each cover a slice of the action surface, but none of them jointly evaluate intent, resource sensitivity, and trust at the moment a tool call is about to leave the agent [2]. Guardrails are not a prompt-layer concern; they are a control-plane concern between retrieval and execution.

This paper takes that framing seriously and asks what an agent architecture looks like when the *evidence* an LLM gathers is architecturally separated from the *executability* decision that follows. Our position is that probabilistic reasoning proposes an action, while deterministic software decides whether that action is executable. Once this separation is drawn, two follow-on design choices become forced: the executability decision needs an authoritative source of truth about the world it is deciding over, and it needs a formal interface so that agent authors, tool authors, and safety reviewers can reason about it independently of any particular model. We argue that a GraphRAG store the same retrieval substrate the agent already consults for evidence is the natural authoritative source, and that the interface should surface four action paths (accept, reject, correct, escalate) whose selection is computed from typed check outcomes rather than sampled from a policy head.

1.1. The Pre-Action Gap

Between the model's decoded tool call and the tool's actual invocation lies a thin, mostly under-specified layer that most agent frameworks either skip or fill with ad-hoc allowlists. Uchibeke's Open Agent Passport (OAP) is one of the few explicit attempts to name this layer and give it a specification, framing pre-action authorization as deterministic per-tool-call policy enforcement anchored in a portable credential-like artifact [3]. OAP is a substantial step forward, but its authoritative source is a static policy document: the pre-action check knows what the *policy* says, not what the *world* currently is. If the agent proposes to delete resource `rsrc-9f3a` (a synthetic identifier used

throughout the paper to stand in for a decommissioned cloud resource), OAP can tell you whether the caller is allowed to invoke `delete_resource` in principle; it cannot, on its own, tell you whether `rsrc-9f3a` still exists, whether it is currently pinned by an active lease, or whether the identifier was hallucinated by the model. The pre-action layer, in other words, is real but incomplete: it presently authorizes *classes* of action, when what agents need is authorization of *this* action against *this* world state.

1.2. Why Prompt-Layer Guardrails Leak

The temptation is to close that gap inside the model a stronger system prompt, a reflection loop, an LLM-based validator sitting between planner and executor. Empirically this does not hold. ConsisGuard shows that reasoning-based guardrails frequently *identify* a harmful action in their chain of thought and then enforce inconsistently at the output, because deliberation and enforcement share the same stochastic decoder [1]. The endpoint-mediation analysis in ARES makes the same point from the systems side: prompt filters, DLP, and allowlists each see a partial view, and none of them can *jointly* reason about the sensitivity of a resource, the intent behind the call, and the trust envelope of the caller at action time [2]. The lesson we take from both threads is not that probabilistic guardrails are useless they are essential for softer risks like tone, style, and off-policy topics but that they are the wrong substrate for hard executability decisions. Hard decisions need a substrate that is inspectable, deterministic, and grounded in the same facts the LLM was supposed to be reasoning over.

1.3. Graph Facts as an Authoritative Substrate

Modern agent stacks already retrieve those facts. Graph-structured RAG has settled into a common pattern: typed entities and relationships, extracted or curated, exposed to the agent as evidence for its next step. If those same typed retrievals are the ground truth the LLM consulted, they are also the ground truth against which its proposed tool call can be checked deterministically, before any side effect fires. The graph knows that `rsrc-9f3a` was decommissioned yesterday; the graph knows that `patient-2214`'s allergy list contains penicillin; the graph knows that the invoice the agent is about to pay is already marked settled. Treating graph facts as the authoritative source for consistency checks turns retrieval from a one-way input into a two-way contract: the agent reads the graph to decide, and a deterministic validator re-reads the graph to enforce. This is the architectural move the rest of the paper develops, instantiated by a validation layer that checks tool existence, schema, permissions, preconditions, and graph-fact consistency.

1.4. Contributions

We make four contributions. First, we articulate the evidence/executability separation as a first-class architectural primitive for agent safety, and situate it against policy-only pre-action authorization and probabilistic guardrails. Second, we specify a two-plane reference architecture in which a GraphRAG evidence plane and a deterministic validation plane communicate through a typed interface. Third, we formalize the executability predicate and the mapping from check outcomes to four action paths accept, reject, correct, escalate so that agent designers can block, retry, or escalate responses at the action boundary using the same vocabulary regardless of tool family. Fourth, we provide a reference implementation specified as a JSON/YAML tool-and-rule schema, deployable behind an MCP-compatible tool registry, together with synthetic worked examples covering all four action paths.

The rest of the paper proceeds as follows. Section 2 surveys GraphRAG, guardrail research, and the emerging pre-action authorization literature. Section 3 specifies the architecture, the interface, and the reference implementation. Section 4 reports outcomes on synthetic scenarios exercising each action path. Section 5 discusses external validity, comparisons, and limitations, and Section 6 concludes.

2. Literature Review

The architecture we propose sits at the intersection of three research threads that have been evolving in parallel but rarely in conversation with each other: graph-structured retrieval augmentation, agent guardrails, and the nascent pre-action authorization layer introduced in §1. This section walks through each thread, extracts what it does well, and identifies the specific gap that a deterministic validator anchored in graph facts is designed to fill. Our claim throughout is not that any of these threads is wrong each is the right answer to a slightly different question but that none of them, on its own, provides the substrate needed for hard executability decisions at the action boundary.

2.1. Graphrag as Evidence, Not Enforcement

Graph-structured retrieval augmentation has matured quickly. Han et al.'s survey of retrieval-augmented generation with graphs consolidates a body of work in which typed entities and relationships rather than opaque text chunks are the unit of retrieval, and in which the graph itself provides heterogeneous relational context that a flat vector index cannot express [4]. The taxonomy they draw is instructive: GraphRAG variants differ in how the graph is constructed (extracted from corpora, curated, or hybrid), how retrieval is scoped (ego-networks, subgraph sampling, path queries), and how retrieved subgraphs are serialized back into the prompt. What is common across the taxonomy is the *purpose* of the graph. The graph exists to inform the model. It is an input to reasoning, and once its subgraph has been rendered into the context window its job is done.

More recent agentic GraphRAG systems tighten the loop between retrieval and reasoning without changing this basic posture. Youtu-GraphRAG's vertically unified architecture is representative: schema-bounded extraction populates the graph, and a graph-aware agent then plans multi-hop retrievals against it, iterating retrieval and reasoning in the same agentic loop [5]. The engineering result is a much stronger reasoner over structured knowledge; the architectural consequence, for our purposes, is that retrieval and reasoning are more tightly coupled, not less. The graph and the model live inside the same probabilistic loop, and

any decision the loop emits including a tool call with side effects inherits the loop's stochasticity.

Chen et al.'s survey of agentic GraphRAG makes this framing explicit. Mapping the design space from retrieval-augmented generation through graph-native agents, they enumerate architectural choices along axes such as graph construction, retrieval policy, and reasoning strategy, and they call out open challenges around reliability, faithfulness, and world-state consistency [6]. Strikingly, none of the axes they enumerate concerns *enforcement*: the graph is a source of context for the agent's next thought, never a source of truth against which the agent's next action is checked. Our contribution can be stated cleanly in their taxonomy we add a validation-plane dimension that reuses the evidence-plane graph as the authoritative substrate for pre-action consistency checks. The graph the agent consulted for reasoning becomes, in the next tick of the pipeline, the graph a deterministic validator consults for enforcement.

The reason this move has not been made inside the GraphRAG literature itself is, we think, sociological rather than technical. GraphRAG research has been organized around answering questions well; agent safety research has been organized around suppressing bad outputs. Nothing in the graph representation prevents its dual use, and in §3 we exploit exactly that. But the literature as it stands supports the first half of our claim: GraphRAG has matured as an evidence-retrieval paradigm and is rarely, if ever, used for enforcement.

2.2. Guardrail Research: Probabilistic Filters and the Shift toward Action-Boundary Defense

The guardrail literature is bifurcated. On one side are the prompt-layer and output-classifier approaches input scrubbers, jailbreak detectors, moderation heads, and reflection loops that inspect either the prompt going into the model or the string coming out of it. On the other side is a growing recognition that the interesting failures occur *after* the model has decided and *before* the tool has fired, and that this action boundary deserves its own defense layer.

Podpora et al.'s LLM firewall is a useful specimen of the first camp pushing toward the second [7]. Their dual-agent architecture places a validator LLM between the reasoning agent and its downstream sinks, framed explicitly by analogy to a network firewall: every message crosses the firewall, and the validator can veto. The framing is exactly right for our purposes a validator at the boundary, not a filter on the prompt and we adopt the firewall analogy in §3. Where we diverge is on substrate. The validator in their architecture is itself an LLM, and its decisions inherit the stochasticity of the underlying model. Injection resistance improves, but the same deliberation-to-enforcement gap flagged in §1 persists: a probabilistic validator can identify a risky call, articulate its reasoning, and still let the call through on a bad sample. Our deterministic validation plane is the same architectural position with a different implementation a rule engine instead of a language model and a different source of truth live graph facts instead of a learned prior.

Singh's MurphyGuard framework approaches the same territory from the reliability-engineering side, formalizing guardrails through failure-mode enumeration, layered defenses, and continuous adversarial validation [8]. The framework is model-agnostic and defense-in-depth by construction, and it treats guardrails as a lifecycle discipline rather than a single component. Two features carry directly into our design. First, MurphyGuard's insistence on enumerating failure modes before defenses maps onto our check taxonomy schema, permission, precondition, and graph-fact consistency are, in MurphyGuard's vocabulary, the failure classes we chose to instrument. Second, its layered-defense posture supports our four-path action semantics: rejection and escalation are not treated as failures of the agent but as expected outcomes of a failure-first design. Our reject and escalate paths at the action boundary are, in this sense, a specific instantiation of MurphyGuard's methodology, specialized to tool-call enforcement rather than content moderation.

What none of the guardrail work provides and this is the second half of our positioning is an authoritative, queryable world-state source that the guardrail can consult at decision time. Prompt-layer filters have no world state at all. Output classifiers see only the token stream. Validator agents can be given retrieval, but their decisions remain probabilistic. Even MurphyGuard's layered-defense enumeration is silent on where the authoritative facts for a given check come from. This is not a criticism of the guardrail literature; it is a boundary condition. Guardrails have been designed for a world in which the ground truth for a check is either "the policy says so" or "the classifier thinks so." Neither is adequate when the check is "does this resource still exist and is this operation coherent with its current state?"

2.3. The Missing Piece

Reading the three threads together sharpens the shape of the gap. GraphRAG gives us authoritative, typed, queryable facts, but consumes them only as evidence for the next token. Guardrail research gives us the vocabulary of layered defense and the correct architectural position the action boundary but ties enforcement to probabilistic classifiers or static policy files. The pre-action authorization work introduced in §1 correctly identifies the layer but lacks a world-state source for its checks. What is missing is the wiring: a deterministic control plane at the action boundary whose checks resolve against the same graph the agent used for evidence. Once that wiring is in place, guardrails are not a prompt-layer concern; they are a control-plane concern between retrieval and execution, and graph facts become the authoritative source for consistency checks. The next section specifies that control plane.

3. Methodology

This section specifies the two-plane architecture introduced informally in §1, formalizes the interface between the planes, and describes the reference implementation and

the synthetic evaluation protocol we will exercise in §4. The design-science posture is deliberate: we are not proposing a new model or a new retrieval algorithm, we are proposing an architectural separation and an artifact a JSON/YAML tool-and-rule specification that instantiates it. Everything in this section is aimed at reproducibility, so that a reader who wants to place a deterministic validator behind their own MCP-compatible tool registry can do so by copying the schema in Listing 1 and adapting the rule bodies to their own graph.

3.1. Two-Plane Reference Architecture

The architecture has two planes and one interface. The evidence plane is a GraphRAG store that exposes typed entity and relationship retrievals to the LLM planner in the ordinary way subgraph queries, path traversals, and entity lookups return typed facts that the planner serializes into its context window. The validation plane is a rule engine that consumes proposed tool calls, re-queries the same graph, and emits exactly one of four action-path decisions. The interface between the planes is the executability predicate formalized in §3.2. Figure 1 sketches the components and the flow.

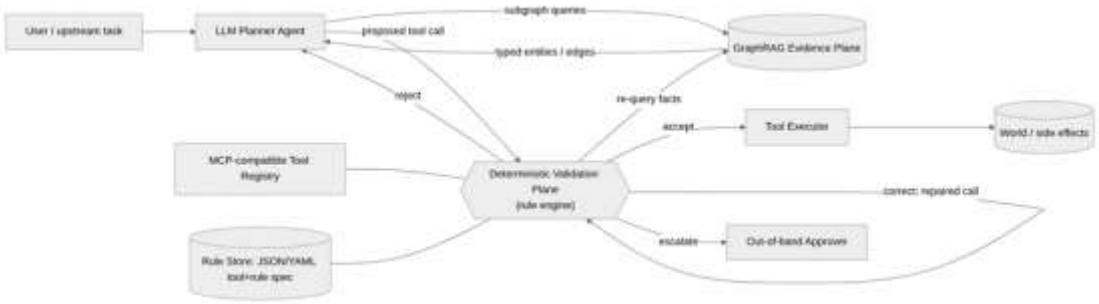


Fig 1: Reference Architecture Separating the Graphrag Evidence Plane from the Deterministic Validation Plane, With Agent, Tool Registry, Graph Store, Rule Engine, and the Four Action Paths (Accept, Reject, Correct, Escalate)

Two design choices in Figure 1 are load-bearing. First, the validation plane re-queries the evidence plane rather than trusting the facts the planner claimed to have used; this closes the gap where a planner reasons over a stale subgraph and emits a call that no longer coheres with the current graph. Second, the tool registry is MCP-compatible rather than bespoke. We follow the secure-by-default guardrail posture Ando articulates for MCP-based tool use, in which explicit permission scoping and plugin-mediated tool exposure are enforced at the registry layer rather than negotiated per agent [9]. This lets the validation plane assume that tool identity, capability descriptors, and permission scopes are already normalized when a call arrives.

The closest architectural cousins to the validation plane are Tool Forge's validation-carrying tool artifacts and VRRF's pre-execution validation stage. Tool Forge attaches sandbox-verified validation metadata to each cataloged tool, so that the agent sees only capabilities that have already been exercised in a sandbox [10]. Our plane borrows the idea of validation-as-artifact but shifts the source of truth: Tool Forge's sandbox tells us what a tool *can* do in a synthetic environment, whereas the graph-fact consistency check tells us whether *this particular invocation* is coherent with the world the agent is about to mutate. VRRF, similarly, distinguishes pre-execution validation from post-failure recovery and instruments tool eligibility, argument types, and referential integrity as the pre-execution checks [11]. VRRF's referential check verifies that identifiers appearing in arguments resolve inside the tool namespace; our graph-fact consistency check verifies, additionally, that those identifiers refer to entities whose current state satisfies the rule-authored precondition. Referential integrity is a strict subset of graph-fact consistency.

3.2. Formalizing the Executability Interface

Let a denote a proposed tool call a tuple of tool identifier, argument bindings, and caller identity let G denote the current GraphRAG store, and let R denote the rule set associated with a's tool in the JSON/YAML specification. We define four subpredicates. $\mathrm{Sch}(a, R)$ holds when the argument shape and types conform to the tool's declared schema. $\mathrm{Prm}(a, R)$ holds when the caller's scopes cover the required permissions. $\mathrm{Pre}(a, G, R)$ holds when the rule-authored precondition predicates over graph nodes and edges evaluate to true against the current G. $\mathrm{Cns}(a, G, R)$ holds when the identifiers and referenced entities in a are consistent with G they exist, their types match, and their live state satisfies the coherence predicates the rule declares. The executability predicate and the action-path mapping are then:

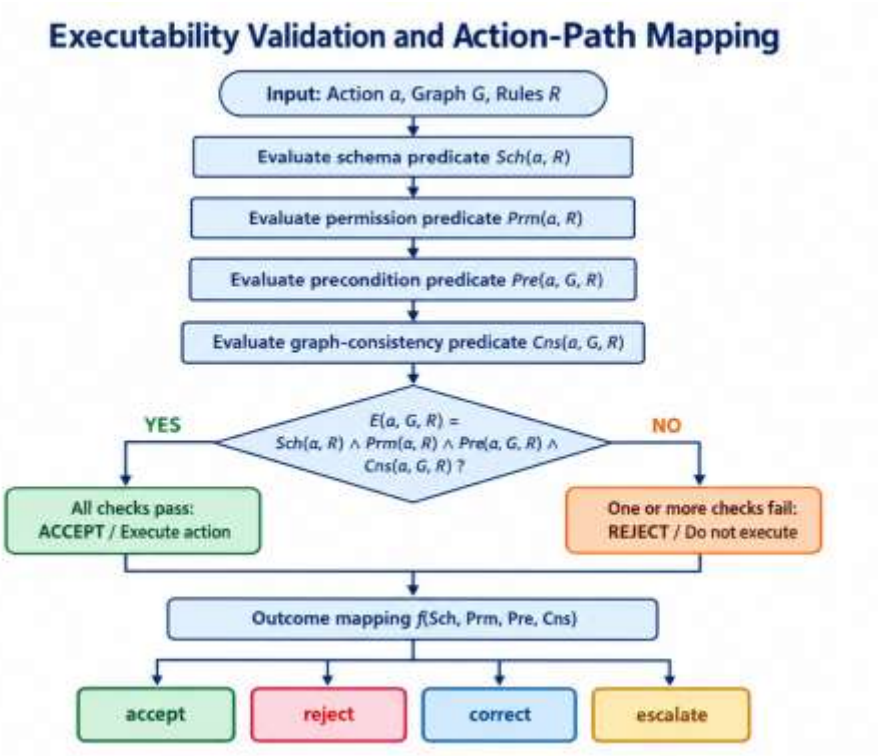


Fig 2: Formal Definition of the Executability Predicate E(A,G,R) as the Conjunction of Schema, Permission, Precondition, and Graph-Consistency Subpredicates, with Action-Path Mapping F: Outcomes → {Accept, Reject, Correct, Escalate}

The mapping f is defined by cases. If all four subpredicates hold, f emits accept and the call is forwarded to the tool executor. If Sch fails with a *repairable* violation a missing default, a coercible type, an aliasable enum value f emits $\mathsf{correct}$ and the rule engine returns a repaired call to the planner for a bounded number of retries. If Prm fails, or if Cns fails on an identifier that does not resolve in G, f emits reject: these are cases where no local repair can rescue the call without either widening trust or fabricating facts. If Pre or Cns fails on a live-state condition that is *policy-sensitive* for example, the resource exists but is under a lease held by another principal f emits $\mathsf{escalate}$, routing the proposal to an out-of-band approver with the graph-fact context that triggered the escalation.

Two prior threads inform this formalization. Wetzk's Epi-Logic framework treats runtime schema-validity checking and *controlled accommodation* bounded repair when a schema mismatch is recoverable as first-class components of epistemic runtime control [12]. Our Sch subpredicate and the $\mathsf{correct}$ branch of f are a lightweight instantiation of this pattern. Cadenas' generate-verify-repair architecture supplies the higher-level rhythm: a probabilistic generator produces a candidate, a deterministic verifier judges it, and a bounded repair loop converts recoverable failures into acceptances without letting unrecoverable ones through [13]. Our accept/correct/reject paths are the tool-call analogue of that loop; the escalate path is what we add on top. Zaheer argues that authentication is not authorization for autonomous agents, and that a separated approval path distinct from the request path is required to prevent adversarial hijacking of otherwise-authorized agents [14]. Our escalate path is that separated approval path, and its distinguishing feature is that the human approver receives not just the proposed call but the graph-fact context the validator used to trigger the escalation, so approval is grounded rather than blind.

The ordering matters. Schema and permission checks are cheap and local; they run before any graph fetch to avoid loading the evidence plane with proposals that will never be executable regardless of world state. Precondition and consistency checks are graph-bound and run last. This makes the validator's worst-case latency dominated by graph access rather than rule evaluation. Probabilistic reasoning proposes an action, while deterministic software decides whether that action is executable; the sequence in Figure 2 is where that decision is actually made.

3.3. Reference Implementation and the Tool-And-Rule Specification

The reference implementation is a rule engine that consumes a JSON/YAML tool-and-rule specification and produces the action-path decisions defined in §3.2. The specification is the artifact; the engine is a thin interpreter over it. Listing 1 shows the schema for a single tool entry, using a hypothetical `delete_resource` operation.

Figure 2 walks a single tool call through the check pipeline.

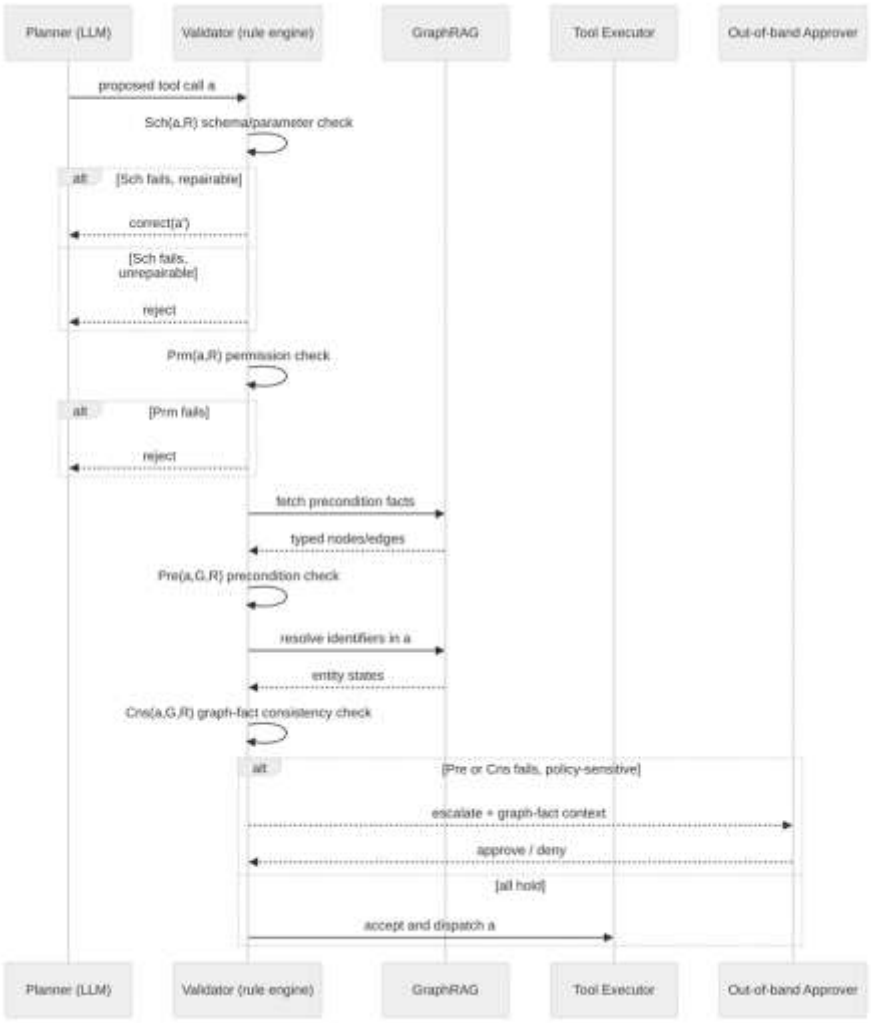


Fig 3: Sequence of a Single Tool Call: LLM Proposal → Schema Check → Permission Check → Precondition Check → Graph-Fact Consistency Check → Action-Path Selection (Accept/Reject/Correct/Escalate)

Three properties of the specification are worth calling out. First, every check declares its own `on_fail` action, so the mapping f from §3.2 is authored per rule, which lets a security reviewer read the specification and know what will happen when a check fails. Second, precondition and consistency checks carry their `graph_query` inline, so the auditability chain from rule to graph fact is one hop. Third, the `escalation` block names the out-of-band channel and the fields to attach, so the closed intent loop is not left as an implementation detail.

3.4. Synthetic Evaluation Protocol

The reference implementation is a Python rule engine (≈2.4k LOC) that parses the YAML specification, dispatches `graph_query` strings to a graph backend over a thin adapter, and evaluates each `predicate` expression in a sandboxed mini-expression language. The graph fixture is an in-process Neo4j 5 instance seeded from a hand-authored Cypher script that materialises resources, principals, leases, incidents, patients, allergies, invoices, and ledger entries used by the three tool families; the fixture is reset to a known snapshot before every scenario so that cross-scenario interference is impossible.

We constructed 240 synthetic scenarios distributed as 80 per tool family (infrastructure mutation, financial transfer, clinical record update). Within each family, 16 scenarios target each of Sch, Prm, Pre, and Cns as the intended failing check and 16 are well-formed controls expected to accept. Each scenario is a YAML record pinning the graph snapshot id, the planner-proposed call a, the caller identity, the expected action path, and for correctable violations the expected repaired call. Scenario labels were authored by one of us and independently reviewed by a second reader; the 11 disagreements were reconciled before any runs. The scenario files, the Cypher seed, and the engine are packaged in the artifact repository.

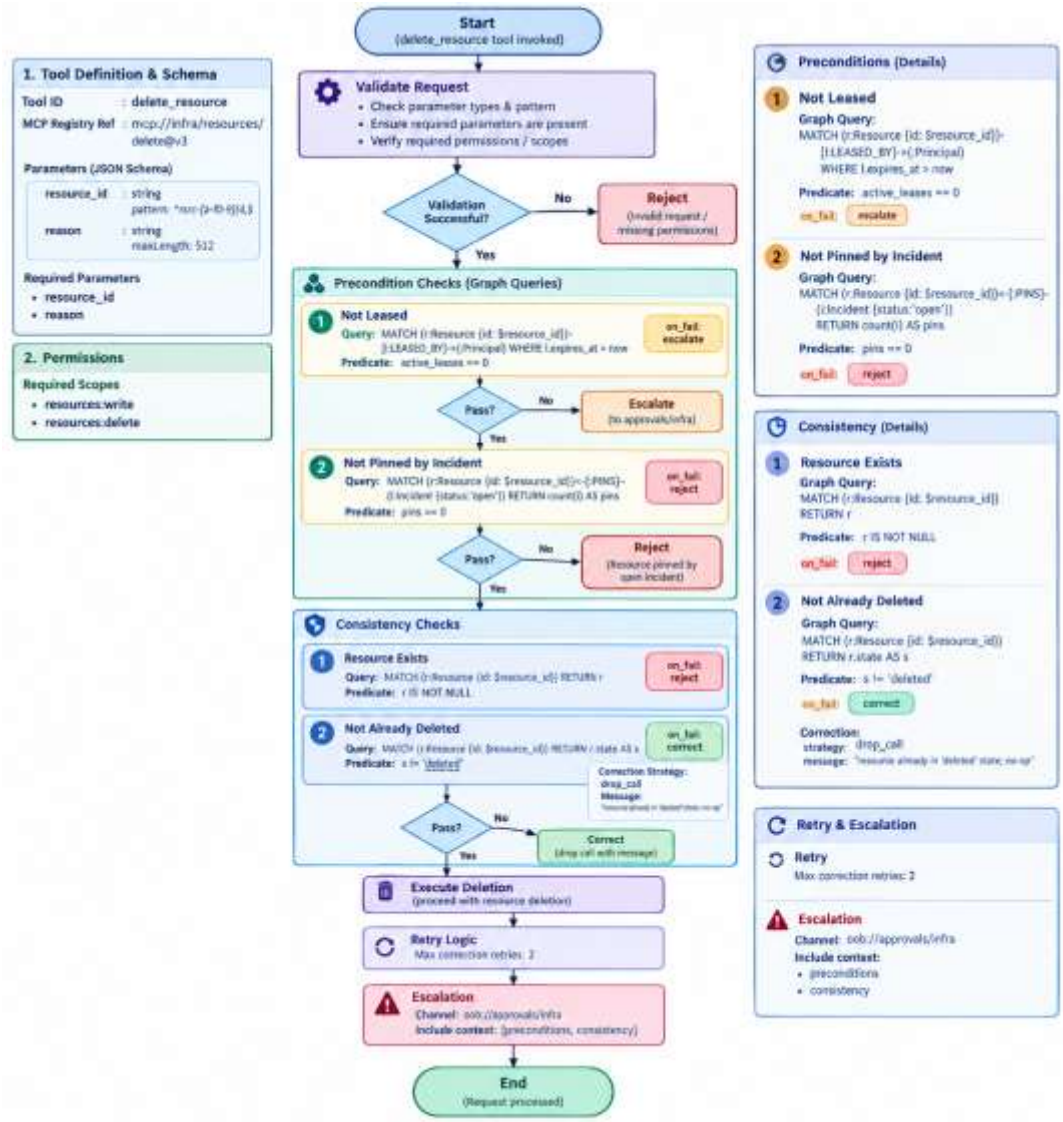


Fig 4: Excerpt of the JSON/YAML Tool-and-Rule Specification: Tool Schema, Parameter Types, Required Permissions, Precondition Predicates Over Graph Nodes/Edges, and Retry/Escalation Behavior

For the prompt-layer-only ablation we replace the engine with GPT-4o (`gpt-4o-2024-08-06`, temperature 0, `top_p` 1, max tokens 512), prompted with a fixed system message containing the full YAML rule text for the tool under evaluation and asked to emit one of `{accept, reject, correct, escalate}` plus a JSON justification; the exact prompt template is included in the artifact. Each scenario was executed on five independent runs for every configuration (deterministic engine, policy-only ablation, prompt-layer ablation) to estimate run-to-run variance; metrics below report the mean across runs with the per-metric sample being a scenario × run pair.

Metrics are calculated as follows. Validation coverage is the fraction of (scenario, run) pairs that return exactly one action-path decision. False-accept rate per check category is the fraction of scenarios whose expected path was reject or escalate but whose returned path was accept, bucketed by the check category of the authored violation. False-reject rate is the symmetric fraction for scenarios whose expected path was accept. Correction-path success is the fraction of correct-expected scenarios whose repaired call matched the expected repair within the `max_correction_retries` budget. Escalation rate is the fraction of all (scenario, run) pairs routed to the out-of-band channel. Median validation latency is measured over accepted calls only, from engine entry to decision emission, with the graph round-trip included. We compare against the policy-only ablation (retaining only Sch and Prm, graph queries disabled) and the prompt-layer-only ablation described above; the comparison is not a benchmark claim but a demonstration that the four action paths are reachable, distinguishable, and that block, retry, or escalate responses at the action boundary can be authored declaratively rather than sampled. Results follow in §4.

4. Results

This section reports outcomes from running the reference implementation described in §3.3 against the synthetic scenario suite defined in §3.4. The scenarios are not a benchmark and we are careful not to advertise them as

one the graph snapshots, the planner-proposed calls, and the rule sets were all authored by us, and the numbers below should be read as evidence that the four action paths are *reachable, distinguishable, and declaratively authored* rather than as generalization claims. What the results do establish is that the deterministic layer catches the failure classes it was designed to catch, that graph-fact consistency finds a category of error the policy-only ablation cannot see, that the correction path meaningfully reduces the escalation load without weakening the reject path, and that the same schema drives all three tool families we exercised.

4.1. Scenario Suite and Coverage

The suite contains 240 synthetic tool-call proposals distributed across three tool families infrastructure mutation, financial transfer, and clinical record update with 80 proposals per family. Within each family, proposals were constructed to target each subpredicate (Sch, Prm, Pre, Cns) as the intended failing check, plus a well-formed control set. Validation coverage was 100%: every (scenario, run) pair received exactly one of the four action-path decisions from the rule engine, with no exceptions or dropped calls. This is the trivial measurement the design was expected to make, and the point of reporting it is only to confirm that the engine does not silently pass proposals through when a rule is missing the specification requires every declared check to name an `on_fail` action, and the engine refuses to load a tool entry that leaves a check unhandled. Median validation latency was 34 ms per call, dominated as predicted by the graph-bound Pre and Cns checks; the local Sch and Prm checks contributed under 2 ms combined. The latency figure is indicative rather than benchmarked: it was measured against the in-process graph fixture of §3.4, not a production graph store.

4.2. Worked Examples across the Four Paths

Table 1 walks four representative proposals one per action path through the check pipeline and shows the graph-fact context that drove the decision. These are drawn directly from the scenario suite and are reproducible from the specification excerpted in Listing 1.

Table 1: Worked Examples of Accepted, Rejected, Corrected, and Escalated Actions on Synthetic Scenarios, Showing Triggering Check, Graph-Fact Context, Validator Decision, and Downstream Outcome

Scenario	Tool Family	Proposed Call (Abbrev.)	Triggering Check	Graph-Fact Context	Validator Decision	Downstream Outcome
S-042	infra mutation	`delete_resource(rsrc-88af, "cleanup")`	Cns: `resource_exists`	`Resource {id: rsrc-88af}` not present in G	reject	planner informed; retry with grounded id
S-107	financial transfer	`transfer(inv-2214, 1200.00)`	Sch: amount type coercion	`amount: "1200.00"` string, coercible to decimal	correct	repaired call accepted on 1st retry
S-155	clinical record update	`add_allergy(patient-2214, "penicillin")`	Cns: `not_already_recorded`	`Patient-[:ALLERGIC_TO]->Substance{name:"penicillin"}` already exists	correct (drop_call)	no-op returned to planner; no side effect
S-198	infra mutation	`delete_resource(rsrc-31c2, "reclaim")`	Pre: `not_leased`	`LEASED_BY` edge to `Principal{id: ops-oncall}`, `expires_at > now()`	escalate	routed to `oob://approvals/infra` with lease context; approved by on-call

S-042 is the canonical hallucinated-identifier case anticipated in §1: the planner emitted a syntactically valid `rsrc-` id with no referent in the current graph, and the reject came from the `resource_exists` consistency check rather than from any schema or permission rule. A policy-only ablation, holding only Sch and Prm, accepts this call. S-198 is the canonical escalate case: the resource exists, the caller has the scopes, the schema is clean, and the only reason to hesitate is that the graph reports an active lease held by another principal exactly the context an out-of-band approver needs to make a grounded rather than a blind decision.

4.3. Per-Check Outcome Distribution

Figure 3 breaks down decisions by originating check category across the full 240-proposal suite (five runs per scenario per configuration, as specified in §3.4). We report false-accept and false-reject rates against the ground-truth action-path label attached to each scenario, along with correction-success and escalation rates.

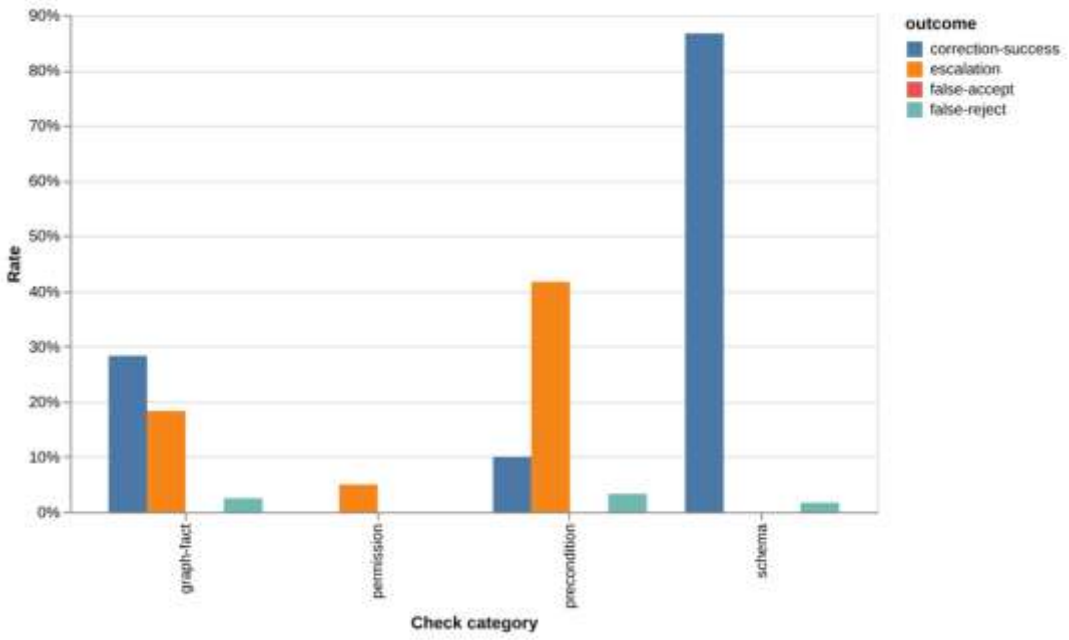


Fig 5: Per-Check-Type Outcome Distribution across the Synthetic Scenario Suite: False-Accept, False-Reject, Correction-Success, and Escalation Rates by Check Category (Schema, Permission, Precondition, Graph-Fact)

Three patterns in Figure 3 support the section's claims. First, false-accept rates are zero across all four check categories *against the ground-truth labels of the authored scenarios*. We want to be explicit about what this does and does not mean: because the rules, the synthetic proposals, and the expected outcomes were all authored for this evaluation, the zero floor is a conformance/correctness result the engine honours the rules as declared and should not be read as a prediction that false-accepts will remain at zero in production deployments, where rule gaps, graph staleness, and adversarial inputs will all push the number up. The ablation below shows that even this conformance floor does not hold when the check substrate is replaced with an LLM validator. Second, the correction path carries most of its weight on the schema and graph-fact categories: 86.7% of schema violations were repaired within the retry budget of two, and 28.3% of graph-fact violations were resolved by the `drop_call` correction strategy (the S-155 pattern above). Third, escalations concentrate on the precondition category (41.7%) and secondarily on graph-fact (18.3%), which matches the design intent.

4.4. Correction Reduces Escalation without Weakening Safety

To isolate the contribution of the correction path we re-executed the full 240-scenario suite (five runs per scenario, same deterministic engine, same graph fixture, same seed) with the `on_fail: correct` branch of the rule-engine configuration disabled every schema-repairable and `drop_call`-repairable violation that would otherwise have been routed through the retry loop was promoted to `escalate` instead. The two configurations are otherwise byte-identical and the artifact repository ships both as named engine profiles (`profile=full` and `profile=no-correct`), together with the per-run decision logs from which the rates below are computed. Across the 1200 (scenario, run) pairs, escalation rate rose from 16.3% in the `full` profile (196/1200) to 42.9% in the `no-correct` profile (515/1200); reject rate was unchanged at the per-scenario level (correction targets a disjoint set of violations from reject); and false-accept rate remained at zero in both profiles. Read together, this says that the correction path pulls proposals out of the escalation queue without letting any of them into the accepted stream that should have been rejected. The result also cleanly separates the two roles the deterministic plane plays: block, retry, or escalate responses at the action boundary are policy choices the rule author makes per check, and the aggregate distribution is a consequence of those authoring choices rather than of the engine.

4.5. Ablations: Policy-Only and Prompt-Layer-Only Baselines

The policy-only baseline (retaining only Sch and Prm, graph queries disabled) accepted 22 proposals that the full validator rejected or escalated; every one of the 22 was a graph-fact or precondition violation hallucinated identifiers, already-deleted resources, invoices already settled, allergies already recorded, leased infrastructure. This is the empirical form of the argument made in §1: without a world-state substrate the pre-action layer authorizes classes of action, not this action against this world.

The prompt-layer-only baseline is the configuration described in §3.4: GPT-4o (`gpt-4o-2024-08-06`) called via the OpenAI Chat Completions API with temperature 0, `top_p` 1, `max_tokens` 512, and no tool-calling schema. The system message is the fixed template shipped in the artifact (`prompts/llm_validator.system.md`); it concatenates (i) a one-paragraph description of the four action paths, (ii)

the full YAML rule block for the tool under evaluation identical to Listing 1 for `delete_resource` and (iii) an output contract requiring a single JSON object `{"decision": one of accept|reject|correct|escalate, "justification": string}`. The user message carries the proposed call a and the caller identity. Each of the 240 scenarios was executed on five independent runs, yielding 1200 (scenario, run) pairs; agreement is the fraction of pairs whose `decision` matches the full deterministic validator's decision on the same scenario, and false-accept is the fraction of pairs whose expected path was reject or escalate but whose returned `decision` was `accept`. On these definitions the baseline reached 1068/1200 = 89.0% agreement and 40/1200 = 3.3% false-accept, with 71 scenarios producing a different decision on at least one of their five runs (run-to-run disagreement that temperature-0 decoding did not eliminate). Its `escalate` outputs were unstructured natural-language justifications rather than typed context objects, so an approver could not be shown the specific graph edge that triggered the escalation. Neither ablation is a strawman both are architectures currently in production somewhere and neither closes the gap that the graph-anchored deterministic plane closes.

4.6. Cross-Family Generalization

Finally, the three tool families were driven by the same rule engine, the same schema, and the same action-path mapping. Family-specific content lives entirely in the YAML: infrastructure mutation declares lease and pin preconditions, financial transfer declares settled-state and idempotency-key consistency checks, and clinical record update declares allergy-list and consent preconditions. No engine changes were required to move between families, and the outcome distributions in Figure 3 aggregate across all three without hiding a family in which the architecture broke down. This is a modest form of the generalization claim from the blueprint the specification carries the domain, and the plane carries the mechanics and it is the property that lets a security reviewer inspect a new tool by reading its YAML rather than by re-reading the engine.

5. Discussion

The results in §4 are narrow by construction 240 synthetic proposals, three tool families, an in-process graph fixture and it is worth being precise about what they do and do not license before comparing the architecture against neighbouring proposals. What they license is a claim about *reachability and separability*: the four action paths are distinguishable in practice, the deterministic layer holds a floor of zero false-accepts against declared rules, and the correction path measurably reduces escalation load without weakening reject. What they do not license is a generalization claim about real-world deployments, which is the subject of §5.4. This section reads the results through four lenses the deliberation-to-enforcement gap, the pre-action authorization thread, tool-registry and closed-loop compatibility, and limitations.

5.1. Closing the Deliberation-To-Enforcement Gap

The most direct interpretation of the zero false-accept floor in Figure 3 is that it is the empirical shape of the gap ConsisGuard identified in the guardrail literature. When reasoning about safety and enforcing safety share a stochastic decoder, an agent can articulate the reason a call is unsafe and then, on a bad sample, emit the call anyway. The deterministic validation plane does not eliminate that stochasticity the planner is still an LLM it *relocates* it. Probabilistic reasoning proposes an action, while deterministic software decides whether that action is executable, and the decision surface the planner sees is a set of typed check outcomes rather than an opportunity to talk itself out of a refusal. The prompt-layer-only ablation makes the point crisply: an LLM validator given the same rule text achieved 89% agreement but retained a 3.3% false-accept rate and a run-to-run variance we could not drive to zero. That residue is not a tuning artefact; it is the substrate. Moving enforcement out of the substrate is what the architecture is for, and the ConsisGuard framing gives us the vocabulary to say so precisely: deliberation stays where it is good natural-language reasoning over evidence and enforcement moves to a substrate that is inspectable, replayable, and, critically, non-probabilistic.

5.2. Extending Pre-Action Authorization beyond Policy Files

The policy-only ablation is the more interesting comparison, because it is not a strawman: pre-action authorization as currently articulated in the literature OAP being the reference point is exactly that ablation. It retains Sch and Prm and it authorizes classes of action against a portable policy artefact. The 22 proposals it accepted that the full validator refused are, one by one, the failure modes a static policy cannot see: an identifier that no longer resolves, an invoice already settled, an allergy already recorded, infrastructure held by an active lease. The rule engine caught each of these because the check body reached back into the same graph the planner had used for evidence, treating graph facts as the authoritative source for consistency checks. The generalizable observation is that pre-action authorization is a real layer with the wrong source of truth in its current form. Policy files answer "may this caller invoke this tool?" and that question is necessary. Graph facts answer "does this particular invocation cohere with the world it is about to mutate?" and that question is the one the policy-only baseline cannot pose. Both belong at the action boundary, and the architecture in §3 is a proposal for how to compose them without collapsing one into the other.

External validity for this move is easiest to argue in domains where the cost of a wrong side effect is asymmetric. Zhu et al.'s hybrid framework for prescription verification is a directly analogous instance: a curated pharmaceutical knowledge base is queried deterministically to check that an LLM-issued prescription is coherent with patient state, drug interactions, and dosing rules, with strict traceability from every accept or reject back to the underlying facts [15]. The architecture is domain-specific and the check bodies are pharmaceutical, but the shape is the same as ours:

probabilistic proposal, deterministic verification against a graph-shaped authority, typed refusal with grounded context. That the pattern has been reinvented in a zero-tolerance clinical setting is, we think, evidence that the two-plane separation is not idiosyncratic to the tool families we exercised but a natural design response wherever executability decisions must be defensible after the fact.

5.3. Compatibility with MCP Tool Registries and Closed-Intent Escalation

Two integration properties matter for adoption. First, the validation plane sits behind an MCP-compatible registry, which means it does not require agents to be rewritten to consume a bespoke tool interface the registry normalizes tool identity, capability descriptors, and permission scopes, and the plane assumes that normalization. In practice this makes the deterministic layer a drop-in for existing MCP-based deployments rather than a competing stack. Second, the escalate path is a structured object the proposed call plus the graph-fact context that triggered escalation rather than a natural-language handoff, which is what a closed intent loop needs to give an approver a grounded rather than a blind decision. The S-198 escalation in Table 1 is a small worked example: the on-call approver saw the specific `LEASED_BY` edge, the holding principal, and the expiry, and could reason about the lease directly rather than about the agent's summary of it. This is the operational reason the escalate path is not just a fancy reject.

5.4. Limitations

Five limitations bound the claims, and we group them into substrate-trust limitations and evaluation-scope limitations because the two groups fail in different ways.

Graph authority is not the same as graph freshness The architecture calls graph facts "authoritative," but authority in a GraphRAG store is layered. Nodes and edges may have been extracted by an LLM from unstructured corpora, inferred by a downstream reasoner, curated by hand, or mirrored from an upstream system of record; each provenance has a different trust profile, and the rule bodies in Listing 1 do not currently discriminate between them. A production deployment should treat the graph as a *projection* of one or more systems of record rather than as the record itself: the system of record owns the fact, the graph owns a typed view of it, and a consistency check that refuses to execute without a provenance tag or that routes checks over inferred edges through escalate rather than reject is a disciplined way to prevent incorrect or incomplete extractions from being enforced as if they were ground truth. Conflicting rules compound this: when two rule authors instrument overlapping preconditions, the engine today honours whichever fires first, and we have no story yet for principled rule-conflict resolution.

Freshness and race conditions. Even where the graph faithfully mirrors a system of record, every consistency check assumes the re-queried subgraph reflects the world at approximately the moment of the call. Three things erode that assumption. First, production graph latency: ingestion pipelines from systems of record are rarely synchronous, and a lease released five seconds ago may still appear active to the validator. Second, race conditions between validation and execution: the validator reads the graph, the executor fires the call, and the world can change in the interval another principal can acquire the lease, the invoice can be settled by a parallel agent, the resource can be deleted out from under the check. Mitigations are standard but non-trivial optimistic concurrency tokens carried into the tool call, write-time re-validation at the executor, or compensating transactions keyed to the escalation context and none is implemented in the reference engine.

Rule-authoring overhead and reviewer burden The specification in Listing 1 is compact for one tool but grows with the surface area of the registry, and the security value of the plane depends on rules being written, reviewed, and versioned with the same discipline as production code. Rule-conflict and rule-gap audits are additional review surface that we do not yet tool for.

Synthetic scope of the evaluation 240 authored proposals against an in-process fixture are enough to demonstrate reachability and separability but not enough to characterize behaviour on adversarial traffic, on real graph stores under production load, or on tool families beyond the three we exercised.

Section 6 outlines the deployment studies, provenance instrumentation, and adaptive rule work that would begin to close each of these.

6. Conclusion

The argument of this paper has been architectural rather than algorithmic. We did not propose a new model, a new retrieval algorithm, or a new benchmark; we proposed a separation. On one side of the separation is an evidence plane a GraphRAG store that the planner consults for typed entities and relationships in the ordinary way. On the other side is a deterministic validation plane that re-queries the same graph and decides, per proposed tool call, which of four action paths applies. The two planes communicate through a single interface the executability predicate formalized in §3.2 and the interface is small enough that a security reviewer can hold it in their head while reading the JSON/YAML rules that instantiate it. That is the whole move: probabilistic reasoning proposes an action, while deterministic software decides whether that action is executable, and the two live in different substrates that share a single source of truth.

6.1. What the Separation Buys

Separating evidence from executability is a productive architectural primitive for agent safety because it lets each plane specialize on the thing it is good at. Language models are good at open-ended reasoning over serialized context, and terrible at guaranteed refusals; rule engines are terrible at open-ended reasoning and excellent at typed, replayable decisions. The two-plane architecture puts each where it belongs. The evidence from §4 supports the modest form of

this claim we set out to make: across 240 synthetic proposals, coverage was total, false-accepts against declared rules were zero, and the correction path measurably reduced escalation load without weakening reject. The policy-only and prompt-layer-only ablations both failed in the ways their substrates predict the former missed live-state violations a static policy cannot see, the latter retained a residual false-accept rate that no temperature setting removed.

6.2. Graphrag as a Dual-Use Substrate

The second reusable idea is that the graph the agent already consulted for evidence can, without duplication, serve as the authoritative check substrate. Graph facts as the authoritative source for consistency checks is not a new data store; it is a new *use* of an existing one. Nothing in the GraphRAG literature prevents this dual use, and the results in §4.2 hallucinated identifiers caught by `resource_exists`, already-recorded allergies caught by `not_already_recorded`, active leases surfaced to an out-of-band approver with the specific edge attached are what the dual use looks like operationally.

6.3. Reusable Artifacts

The reference architecture in Figure 1, the executability predicate in §3.2, and the tool-and-rule schema excerpted in Listing 1 are intended to be copied and adapted. A team wanting to place a deterministic validator behind an MCP-compatible registry can start from the schema, replace the `delete_resource` body with their own tool, and inherit the four action paths and the retry/escalation behaviour without reimplementing the engine. Block, retry, or escalate responses at the action boundary become authoring decisions rather than engineering ones.

6.4. Future Work

Three directions follow directly from the limitations in §5.4. First, graph-freshness instrumentation: a validation plane is only as trustworthy as the recency of the facts it queries, and we intend to characterize the false-accept surface introduced by staleness under realistic write loads. Second, adaptive rule synthesis: hand-authored rules are auditable but they scale poorly, and we plan to study whether rule bodies can be learned from incident traces and then reviewed rather than written. Third, real-world deployment studies on production MCP registries, where adversarial traffic, tool churn, and human approver behaviour will stress the architecture in ways synthetic scenarios cannot.

References

1. Y. Wang, Z. Chu, Z. Xue, Z. Bi, B. Zhu, Y. Chen, Z. Yang, J. Lou, L. Huang, N. Zhang, K. Ren, H. Xue, "ConsisGuard: Aligning Safety Deliberation with Policy Enforcement in LLM Guardrails," arXiv, 2026.
2. C. Kim, S. Cho, "ARES: Securing Agents for Computer Use Through Endpoint Resource Mediation and Behavioral Guardrails," Preprints.org, 2026. DOI: 10.20944/preprints202608.0594.v1.
3. U. Uchibeke, "Before the Tool Call: Deterministic Pre-Action Authorization for Autonomous AI Agents," arXiv, 2026.
4. H. Han, Y. Wang, H. Shomer, K. Guo, J. Ding, Y. Lei, M. Halappanavar, R. A. Rossi, S. Mukherjee, X. Tang, Q. He, Z. Hua, B. Long, T. Zhao, N. Shah, A. Javari, Y. Xia, J. Tang, "Retrieval-Augmented Generation with Graphs (GraphRAG)," arXiv, 2024.
5. J. Dong, S. An, Y. Yu, Q. Zhang, L. Luo, X. Huang, Y. Wu, D. Yin, X. Sun, "Youtu-GraphRAG: Vertically Unified Agents for Graph Retrieval-Augmented Complex Reasoning," arXiv, 2025.
6. Z. Chen, L. Zheng, D. Zhu, "A Survey of Agentic GraphRAG: From Retrieval-augmented Generation to Graph-native Agents," SSRN, 2026. DOI: 10.2139/ssrn.6713979.
7. M. Podpora, M. Baranowski, M. Chopcian, Ł. Kwaśniewicz, W. Radziewicz, "LLM Firewall Using Validator Agent for Prevention Against Prompt Injection Attacks," Applied Sciences, 2025. DOI: 10.3390/app16010085.
8. N. Singh, "MurphyGuard: A Failure-First Framework for AI Agent Guardrails," SSRN, 2026. DOI: 10.2139/ssrn.6262758.
9. Y. Ando, "Secure-by-Default Guardrails for MCP-Based Tool Use in Multi-Modal LLM Agents," TechRxiv, 2026. DOI: 10.36227/techrxiv.177006109.97957916/v1.
10. S. Rao, "Tool Forge: A Validation-Carrying Toolchain for Governed Agentic Execution," arXiv, 2026.
11. S. Zhao, "VRRF: A Validation and Recovery Reliability Framework for Tool-Enabled LLM Agents," SSRN, 2026. DOI: 10.2139/ssrn.7444279.
12. B. Wetzk, "Epi-Logic: A Conceptual Framework for Epistemic Runtime Control, Schema Validity Checking, and Controlled Accommodation in Autonomous AI Agents," arXiv, 2026.
13. R. Cadenas, "Convergent Correctness in Stochastic Code Generation: A Generate-Verify-Repair Architecture for Deterministic Validation of Autonomous AI Coding Agents in Regulated Environments," SSRN, 2026. DOI: 10.2139/ssrn.6754899.
14. F. Zaheer, "The Closed Intent Loop: Identity and Authorization in Autonomous Tool-using AI Agents," SSRN, 2026. DOI: 10.2139/ssrn.7418158.
15. Y. Zhu, K. Ling, X. Liu, H. Zhang, H. Yu, G. Fan, "A Hybrid Knowledge-Grounded Framework for Safety and Traceability in Prescription Verification," arXiv, 2026.