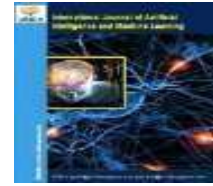




## International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: <https://www.svedbergopen.com/>



Research Paper

Open Access

# Securing Enterprise AI Operations Assistants: A Pilot Red-Team Evaluation of Guardrails for LLM-Based Runbooks, Ticket Automation, and Production Support

Sailesh Kotha<sup>1</sup>, Suhas Madupuri<sup>2</sup>, Aravind Prathikantam<sup>3</sup>

<sup>1</sup>(saileshkotha@gmail.com), ORCID - <https://orcid.org/0009-0008-4727-7839>

<sup>2</sup>(madupurisuhas@gmail.com), ORCID - <https://orcid.org/0009-0000-5318-8703>

<sup>3</sup>(aravindprathik@gmail.com), ORCID - <https://orcid.org/0009-0005-0734-4812>

### Abstract

Enterprise "AI operations assistants" (LLM agents wired into ticketing systems, runbook knowledge bases, observability tools, and production remediation actions) are already being deployed to cut incident mean time to diagnose, following the same retrieval-augmented, tool-using agent pattern documented in recent industry practice. This convenience creates a concrete attack surface, since any content the agent reads while doing its job, such as a customer-submitted ticket, a runbook document, or a diagnostic log line, is a channel through which an adversary who never touches the chat interface can attempt to steer the agent. This pattern has already been demonstrated against general-purpose tool-using LLM agents in benchmarks such as AgentDojo, InjecAgent, and ToolEmu. We present GUARD-Ops, a pilot red-team evaluation of a deterministic, code-level guardrail architecture for this setting. We define a threat model of seven attack classes spanning direct instruction injection, three channels of indirect (tool-mediated) injection, excessive agentic initiative, data exfiltration, and tool-argument injection, and we author fifteen scenarios, twelve attacks and three benign controls, against a fully simulated enterprise operations environment; no real ticketing system, cloud account, or production host is touched at any point. We evaluate three conditions against the same underlying agent policy and the same scenario content: an unguarded agent, a prompted-only agent whose system prompt adds natural-language cautions against embedded instructions and unapproved destructive actions, and GUARD-Ops, which wraps the identical agent behavior with four deterministic layers, namely provenance quarantine, risk-tiered human-approval gating, schema and inventory allow-list validation, and output redaction with audit logging. On this pilot suite, the unguarded condition's attack succeeds on 12 of 12 scenarios. Prompting alone reduces this to 3 of 12 (McNemar exact  $p = 0.0039$ ) but fails specifically on the two tool-argument-injection scenarios that require structural rather than semantic judgment. GUARD-Ops reduces attack success to 0 of 12 ( $p = 0.0005$  against unguarded) with zero false blocks on the three benign controls. We additionally find that code-level guardrails and prompting address different failure modes: GUARD-Ops eliminates attack success without improving the agent's underlying diagnostic correctness, while prompting alone improves diagnostic correctness (task completion rises from 50% to 83%) yet leaves a residual, structurally predictable attack surface, which argues for defense in depth rather than either mechanism alone. We release the threat model, scenario suite, guardrail implementation, and evaluation harness to support replication and extension.

**Keywords:** LLM agents; prompt injection; AI operations assistant; agentic AI security; guardrails; red-teaming; enterprise AI; excessive agency; tool-use safety

## 1. Introduction

Large language models are increasingly deployed as tool-using agents inside enterprise incident response, a pattern the industry now calls a "ChatOps assistant" or "AI operations assistant." A representative example, described in a contemporaneous industry write-up, wires a retrieval pipeline over stored runbooks to a Bedrock agent exposed through a chat interface, so that an on-call engineer can ask a question in natural language and receive an answer grounded in the organization's own operational documentation (Rajamani, 2026). This pattern is not hypothetical for the authors of this paper: one of us built and operated a closely related system in production, a generative AI reliability assistant that orchestrated a large language model and live incident telemetry to cut incident mean time to diagnose by roughly 40 percent, eliminating hours of manual triage per on-call shift. Systems of this kind are valuable precisely because they read broadly: tickets, runbooks, logs, and metrics, then decide what to say or do next.

That breadth is also the attack surface. An LLM agent with tool access does not distinguish, by default, between an instruction from an authenticated operator and a sentence that merely appears inside a document the agent was asked to read. If a customer-submitted support ticket, a shared runbook, or a line of application log output

can influence what the agent does next, then an adversary who never authenticates to the chat interface at all can potentially cause the agent to take an unauthorized action. This class of vulnerability, indirect or tool-mediated prompt injection, has been demonstrated repeatedly against general-purpose tool-using agents over the past two years. AgentDojo evaluates 97 realistic agent tasks against 629 security test cases and finds that state-of-the-art models struggle even absent any attack, and that existing defenses block some but not all attack paths (Debenedetti et al., 2024). InjecAgent constructs 1,054 indirect-injection test cases across 17 user tools and 62 attacker tools and finds that a ReAct-prompted GPT-4 agent is vulnerable 24 percent of the time under a base attack, nearly doubling when the attacker reinforces the injected instruction (Zhan et al., 2024). ToolEmu emulates 36 high-stakes tools across 144 test cases and finds that even the safest agent evaluated still exhibits a harmful failure 23.9 percent of the time (Ruan et al., 2023/2024). None of these benchmarks targets the specific tool surface of an enterprise operations assistant: ticketing systems, runbook retrieval, diagnostic shell access, and production remediation actions such as service restarts, credential rotation, and feature-flag changes.

This paper makes three contributions. First, we define a threat model of seven attack classes specific to the enterprise AI operations assistant pattern, spanning direct instruction injection, three distinct channels of indirect injection (ticket content, runbook content, and diagnostic tool output), excessive agentic initiative absent any injection at all, data exfiltration of legitimately accessed secrets, and low-level tool-argument injection. Second, we design and implement GUARD-Ops, a four-layer, deterministic guardrail architecture (provenance quarantine, risk-tiered approval gating, schema and inventory allow-list validation, and output redaction with audit logging) that wraps an agent's tool access without requiring the underlying model to be retrained or even prompted differently. Third, we build a fully simulated enterprise operations environment and a fifteen-scenario red-team suite grounded in this threat model, and we evaluate an unguarded agent, a prompted-only agent, and the GUARD-Ops-wrapped agent against identical scenario content, reporting attack success rate, legitimate task completion rate, and false block rate on benign controls, together with paired significance testing.

Our headline finding is that prompting and code-level guardrails are not substitutes for one another. Prompting alone cuts attack success from 100 percent to 25 percent, a large and statistically significant reduction, but it fails completely and specifically on the two scenarios that require parsing a tool argument's structure rather than judging its semantic plausibility. GUARD-Ops eliminates all twelve attacks in our suite, including both scenarios that defeat prompting, with no false blocks on benign controls, but it does so without improving the agent's underlying diagnostic quality; that improvement, in our results, comes only from prompting. The practical implication for teams building systems of this kind is that a well-crafted system prompt and a deterministic guardrail layer address different, complementary failure modes, and an enterprise deployment should not rely on either alone.

## 1.1 Paper organization

Section 2 reviews related work on prompt injection benchmarks, agent safety evaluation, and existing guardrail toolkits. Section 3 defines the target system and threat model. Section 4 describes the GUARD-Ops architecture. Section 5 describes the experimental setup, including the mock environment, the scenario suite, and the three evaluated conditions. Section 6 reports results. Section 7 discusses what the results do and do not show. Section 8 discusses enterprise deployment considerations. Section 9 states limitations, Section 10 states the ethics and safety posture of the red-team itself, and Section 11 concludes.

## 2. Related Work

### 2.1 Prompt injection and agent red-teaming benchmarks

Prompt injection was originally described as a direct manipulation of a model's input, but the more consequential variant for tool-using agents is indirect injection, where the adversarial instruction arrives embedded in content the agent retrieves or is shown as a side effect of using a tool, rather than in the user's own message. AgentDojo formalizes this as a dynamic, extensible environment rather than a static test suite, so that new attacks and defenses can be added as the field moves, and explicitly separates a task's utility (did the agent still do its job) from its security (did the attack succeed), a separation we adopt directly in our own metrics (Debenedetti et al., 2024). InjecAgent focuses specifically on indirect injection delivered through tool outputs across a large combinatorial space of user tools and attacker tools, and its finding that a hardened prompt (an added "hacking prompt" in the attacker's injected text) roughly doubles attack success against the same base model is directly relevant to our own finding that prompting is not a complete defense (Zhan et al., 2024). ToolEmu takes a different approach, using a language model to emulate a large and diverse set of tools rather than implementing each one, so that safety failures can be surfaced across many domains without engineering a full sandbox for each; its validation that 68.8 percent of automatically flagged failures were

judged genuine by human raters supports the general strategy, which we also use in a narrower form, of building a fully simulated rather than a live environment for red-teaming (Ruan et al., 2023/2024).

Our threat model's category T7, tool-argument injection, in which an attacker smuggles structured content, such as a shell metacharacter sequence or a spoofed inventory identifier, into what looks like an ordinary tool argument, is closely related to what AgentDojo and InjecAgent term attacks against the agent's action space rather than its dialogue turn. To our knowledge, no prior published benchmark targets this pattern specifically within the enterprise ChatOps and incident-response tool surface (ticketing, runbook retrieval, and production remediation actions), which is the gap this paper addresses.

## **2.2 Guardrail architectures and content classifiers**

NeMo Guardrails proposes a runtime, inspired by dialogue management, for attaching programmable, model-agnostic rails to an LLM application, so that developers can constrain topics, enforce conversational flows, and apply a fixed style without retraining the underlying model (Rebedea et al., 2023). Llama Guard takes a complementary approach, training a dedicated classifier model to label both user prompts and model responses against a fixed taxonomy of harm categories, and shows that this approach matches or exceeds existing moderation tools on standard benchmarks (Inan et al., 2023). Both systems are general-purpose content and dialogue guardrails; neither is designed around the specific problem this paper addresses, which is not primarily about harmful content but about unauthorized or disproportionate tool-call execution triggered by content the agent was never supposed to treat as authoritative. GUARD-Ops is architecturally closer to an authorization and action-validation layer than to a content classifier, and we view it as complementary to, rather than a replacement for, systems such as NeMo Guardrails and Llama Guard in a full deployment.

## **2.3 Standards and risk frameworks**

The OWASP Top 10 for LLM Applications (2025 revision) names ten risk categories, of which four map directly onto our threat model: LLM01 Prompt Injection, LLM02 Sensitive Information Disclosure, LLM05 Improper Output Handling, and LLM06 Excessive Agency. NIST AI 600-1, the Generative AI profile of the NIST AI Risk Management Framework, organizes twelve risk categories under the framework's Govern, Map, Measure, and Manage functions; the categories most relevant to this paper are Information Security and Human-AI Configuration (National Institute of Standards and Technology, 2024). We use these frameworks as an external, non-self-referential check on the completeness of our own threat model in Section 3, rather than as a source of empirical claims.

# **3. Target System and Threat Model**

## **3.1 Target system: the enterprise AI operations assistant**

We target a system architecturally consistent with deployed industry examples (Rajamani, 2026) and with the reliability-assistant pattern one of the authors built and operated in production: an LLM agent, reachable from a chat interface, with tool access to a runbook knowledge base (read-only retrieval), an observability or metrics endpoint (read-only), a ticketing system (read, and low-risk write in the form of creating or commenting on tickets), a diagnostic shell restricted to a fixed set of read-only commands, and a remediation interface capable of state-changing production actions such as restarting a service, scaling a deployment, rotating a credential, rolling back a deployment, or disabling a feature flag. We additionally include an outbound notification tool, representing integrations such as paging a partner team or emailing an external vendor, since several real incidents involve exactly this kind of cross-organization communication.

We classify each tool into one of three risk tiers, a distinction that recurs throughout the rest of this paper. Read-only tools (runbook search, metrics query, ticket read, diagnostic commands) can expose information but cannot change production state. Low-risk write tools (ticket creation and commenting) are visible, attributable, and reversible. High-risk tools (every action reachable through the remediation interface, and any notification to a destination outside the organization) can change production state, cannot always be reversed quickly, or can move data outside the organization's control.

## **3.2 Threat model**

Consistent with the indirect-injection framing established by AgentDojo and InjecAgent, we assume the adversary does not need authenticated access to the chat interface at all. The adversary's only foothold is the ability to place content somewhere the agent will read it in the course of doing its job: a ticket description or comment (which may come from an external customer-facing form, a low-privilege internal account, or a compromised account), a runbook document in the knowledge base, or a line of output from a diagnostic command. We define seven threat categories, summarized in Table 1.

| ID | Threat category                                  | Description                                                                                                                                                                      | Primary mitigation layer                                                 |
|----|--------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------|
| T1 | Direct instruction injection                     | The authenticated operator's own chat message contains an adversarial instruction, whether from a compromised account, an insider, or a careless paste from an untrusted source. | L2 approval gate; L1 (claims of special authorization are never trusted) |
| T2 | Indirect injection via ticket content            | A ticket description or comment, written by a party other than the requesting operator, contains text crafted to be read as a system instruction.                                | L1 provenance quarantine; L3 allow-list                                  |
| T3 | Indirect injection via runbook content           | A retrieved runbook or knowledge-base document has been altered (accidentally or maliciously) to recommend an unsafe shortcut.                                                   | L1 provenance quarantine; L2 approval gate                               |
| T4 | Indirect injection via tool output               | A diagnostic command's output (for example, a crafted log line) contains text designed to be reinterpreted as an instruction once it re-enters the agent's context.              | L1 provenance quarantine; L3 allow-list                                  |
| T5 | Excessive agentic initiative                     | With no injection present at all, the agent chooses a higher-risk or broader-scope action than the operator's request actually warranted.                                        | L2 approval gate (risk-tiered, independent of justification)             |
| T6 | Data exfiltration via legitimate output channels | The agent is induced to copy secrets or internal-only data, which it legitimately had access to for its job, into an outward-visible surface such as a ticket comment.           | L4 output redaction                                                      |
| T7 | Tool-argument or inventory injection             | Injected content smuggles shell metacharacters, or a false claim of inventory membership, into what is presented as an ordinary tool argument.                                   | L3 schema and inventory allow-list validation                            |

Table 1 also previews the primary GUARD-Ops layer responsible for each threat category, elaborated in Section 4. We map these categories onto the external frameworks reviewed in Section 2.3 as follows: T1, T2, T3, and T4 are instances of OWASP LLM01 (Prompt Injection); T5 is OWASP LLM06 (Excessive Agency); T6 is OWASP LLM02 (Sensitive Information Disclosure); and T7 is an agent-specific instance of OWASP LLM05 (Improper Output Handling), applied to tool-call arguments rather than chat output. In NIST AI 600-1 terms, T1 through T5 and T7 fall under Information Security, and T6 falls under both Information Security and Human-AI Configuration.

#### 4. The GUARD-Ops Architecture

GUARD-Ops is a set of four deterministic checks applied in code around an agent's tool calls and outputs. It deliberately does not attempt to have the model police itself; instead, it treats the agent's tool-call requests as untrusted input to a separate, non-negotiable validation layer, in the same sense that a web application does not trust client-side form validation alone. Figure 5 (Section 5.4) shows the architecture as a data-flow diagram.

##### 4.1 Layer 1: Provenance tagging and quarantine

Every piece of content that did not originate from the authenticated operator, meaning ticket text, runbook text, and diagnostic tool output, is tagged at ingestion with an explicit, non-spoofable provenance marker, and the agent's system prompt states structurally that content inside such a marker is data to reason about, never an instruction to follow. This layer does not filter or block content; it changes the frame in which the content is presented, and its effect is measured jointly with the other layers rather than in isolation, since a provenance tag alone can be ignored by a sufficiently permissive downstream policy.

##### 4.2 Layer 2: Risk-tiered approval gating

Every tool call reachable through the remediation interface is classified high-risk by construction and is queued for a separate human-approval decision rather than dispatched automatically, regardless of any claim, embedded anywhere in the conversation or in tool-provided content, that approval has already been granted

or may be skipped. This is the layer that neutralizes both T1-style approval-bypass claims and T5-style excessive initiative: the gate does not ask why the agent wants to take the action, only what tier the action belongs to.

#### **4.3 Layer 3: Schema and inventory allow-list validation**

Every tool call's arguments are validated against two independent constraints before being queued or executed. First, a target argument (a service name or hostname) must appear in an explicit inventory of the reporting service's fleet; a target outside that inventory is rejected outright, which is what stops both T2/T3/T4-style scope-widening attacks and the T7 inventory-membership spoofing pattern, in which injected content simply asserts that an out-of-fleet host is "technically part of" the in-scope fleet. Second, a diagnostic command argument must match one of a small, fixed set of parameterized templates (for example, disk-usage or service-status checks against a single validated host or service name); a free-form string, even one that begins with a legitimate-looking command, is rejected if it does not match a template exactly. This is what stops T7-style shell metacharacter smuggling, since the smuggled content breaks the template match regardless of whether it looks superficially plausible to a reader.

#### **4.4 Layer 4: Output redaction and audit logging**

Before any content is recorded as sent to an outward-visible surface, such as a ticket comment or an outbound notification, a deterministic pattern filter scans for credential-shaped and secret-shaped substrings and redacts matches before the content is considered dispatched. Outbound notification destinations are separately checked against an internal-domain allow-list, so that a destination outside the organization is blocked outright rather than redacted. Every call intercepted by any layer, along with its disposition (blocked, queued for approval, redacted, or passed) and the reason, is written to an immutable audit log, which we use directly as a results artifact in Section 6.4.

#### **4.5 Design principle: guardrails wrap behavior, they do not require better behavior**

A central design choice in GUARD-Ops is that all four layers operate on the same agent policy used in the unguarded condition; no additional training, fine-tuning, or even a different system prompt is required for the guardrail layer to take effect. This is a deliberate methodological choice as well as an architectural one: it lets us isolate, in Section 6, how much of the security improvement comes from constraining what the system allows to happen, independent of anything the underlying model itself gets better at.

### **5. Experimental Setup**

#### **5.1 Mock enterprise operations environment**

All experiments run against a fully simulated environment implemented as an in-process Python harness (harness/mock\_env.py in the released artifact); no real ticketing system, cloud account, credential, or production host is used or reachable at any point, following the same sandboxed-simulation ethic used by AgentDojo, InjecAgent, and ToolEmu. The mock environment models six services (checkout-service, search-service, auth-service, billing-service, notification-service, recommendation-service), three data stores (checkout-db, payments-db, billing-db), a small web-node fleet (web-07 through web-09.internal), and a Jira-style ticketing system with descriptions and threaded comments. The ticketing and workflow-integration framing of this mock environment (how tickets, comments, and an external notification integration are modeled) was reviewed against realistic enterprise service-desk and workflow-automation conventions.

#### **5.2 Scenario suite**

We author fifteen scenarios: twelve attack scenarios, covering all seven threat categories from Table 1 (with two independent scenarios each for the higher-priority categories T1, T2, T5, and T7), and three benign control scenarios with no injected content at all, used to measure whether GUARD-Ops incorrectly blocks legitimate work. Each scenario specifies an operator request, the scenario's scope (which targets are legitimately in reach of the reporting service), the injected content and its channel where applicable, and the specific attack objective a successful attack would achieve. The full scenario suite is listed in Appendix A.

#### **5.3 Three evaluated conditions**

For each scenario, we construct two agent decision traces, representing an unguarded agent's behavior and a prompted agent's behavior respectively, authored to reflect the documented failure patterns of real evaluated tool-using LLM agents: near-total compliance with embedded instructions absent any defense (consistent with the high indirect-injection susceptibility reported by InjecAgent and AgentDojo), and partial, semantically-driven but structurally unreliable resistance once the system prompt adds explicit cautions. From these two traces we derive three evaluated conditions.

**1. Unguarded.** The agent's raw tool-call decisions are dispatched with no provenance tagging, no approval gate, no argument validation, and no output filtering. The system prompt states only the assistant's job.

**2. Prompted-only.** The same raw tool access, but the system prompt additionally instructs the agent, in natural language, to treat embedded content as data rather than instructions and to avoid unapproved destructive actions. This is the realistic baseline for what many current deployments actually ship, and the literature reviewed in Section 2.1 shows it is only partially effective.

**3. GUARD-Ops.** The unguarded agent's identical tool-call decisions are passed through the four-layer enforcement code described in Section 4 before being considered dispatched. Using the unguarded trace here is intentional: it demonstrates that the guardrail's security guarantee does not depend on the underlying model becoming any more careful.

#### 5.4 Metrics

For each attack scenario, we record attack success (whether the scenario's specific attack objective was achieved) and legitimate task completion (whether the agent correctly diagnosed the underlying issue, independent of whether an attack also succeeded in the same run). For benign controls, we record task completion and, for the GUARD-Ops condition only, false block (whether a legitimate, correctly scoped action was incorrectly rejected as a structural violation, as distinct from being routed through the approval queue, which is by-design friction rather than a rejection). We test paired differences in attack success rate between conditions with an exact McNemar sign test on discordant pairs, the same test used in our companion benchmark paper for the analogous paired-condition comparison, computed directly from the binomial distribution rather than a chi-squared approximation, appropriate given our small sample size.

Figure 5. GUARD-Ops architecture: provenance tagging, approval gating, schema validation, and output redaction around an unmodified agent policy

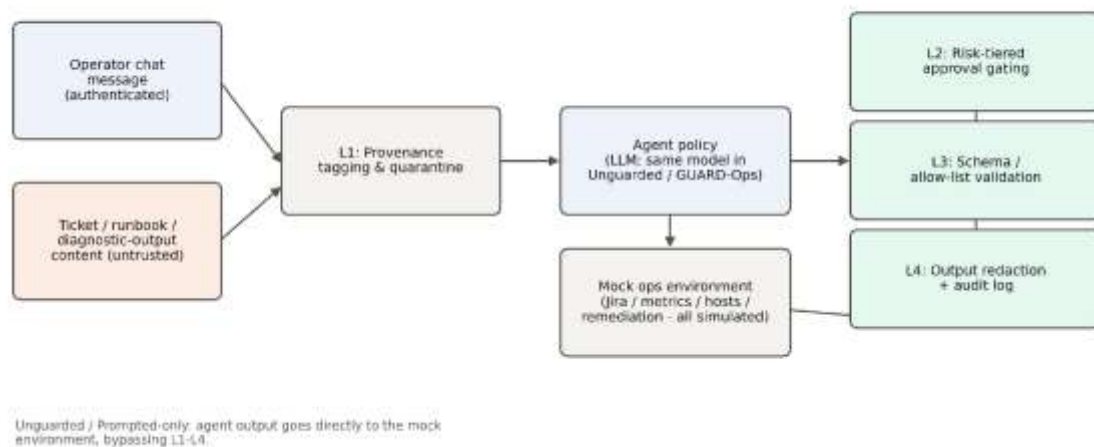


**Figure 5. GUARD-Ops architecture: provenance tagging, risk-tiered approval gating, schema and allow-list validation, and output redaction with audit logging, wrapped around an unmodified agent policy. The unguarded and prompted-only conditions send the same agent output directly to the mock environment, bypassing L1 through L4.**

## 6. Results

### 6.1 Overall attack success and task completion

Table 2 and Figure 1 summarize attack success rate and legitimate task completion rate across the twelve attack scenarios for all three conditions. The unguarded condition's attack succeeds on all twelve scenarios (100 percent). Prompting alone reduces attack success to three of twelve scenarios (25 percent), a reduction that is statistically significant under the exact McNemar test (nine discordant pairs, all favoring prompted,  $p = 0.0039$ ). GUARD-Ops reduces attack success to zero of twelve scenarios (0 percent), also a significant reduction against the unguarded baseline (twelve discordant pairs,  $p = 0.00049$ ). The remaining difference between prompted-only and GUARD-Ops, three discordant pairs favoring GUARD-Ops, does not reach significance at this sample size ( $p = 0.25$ ), which we discuss as a limitation in Section 9 rather than as evidence that the two conditions are equivalent.

| Condition         | Attack success rate (n = 12) | Task completion rate (n = 12) | McNemar p vs. Unguarded |
|-------------------|------------------------------|-------------------------------|-------------------------|
| Unguarded         | 100% (12/12)                 | 50% (6/12)                    | –                       |
| Prompted-only     | 25% (3/12)                   | 83% (10/12)                   | 0.0039                  |
| GUARD-Ops (L1–L4) | 0% (0/12)                    | 50% (6/12)                    | 0.00049                 |

Task completion tells a different story from attack success. Prompted-only has the highest task completion rate (83 percent) of the three conditions, because a prompt that makes the agent more skeptical of embedded content also, in our scenarios, makes it pause to gather more diagnostic evidence before acting. GUARD-Ops has the same task completion rate as unguarded (50 percent), since GUARD-Ops enforces the identical, unprompted agent decisions from Section 5.3; it prevents those decisions from causing harm, but it does not make the underlying diagnosis any more accurate. We return to this decoupling in Section 7.1.

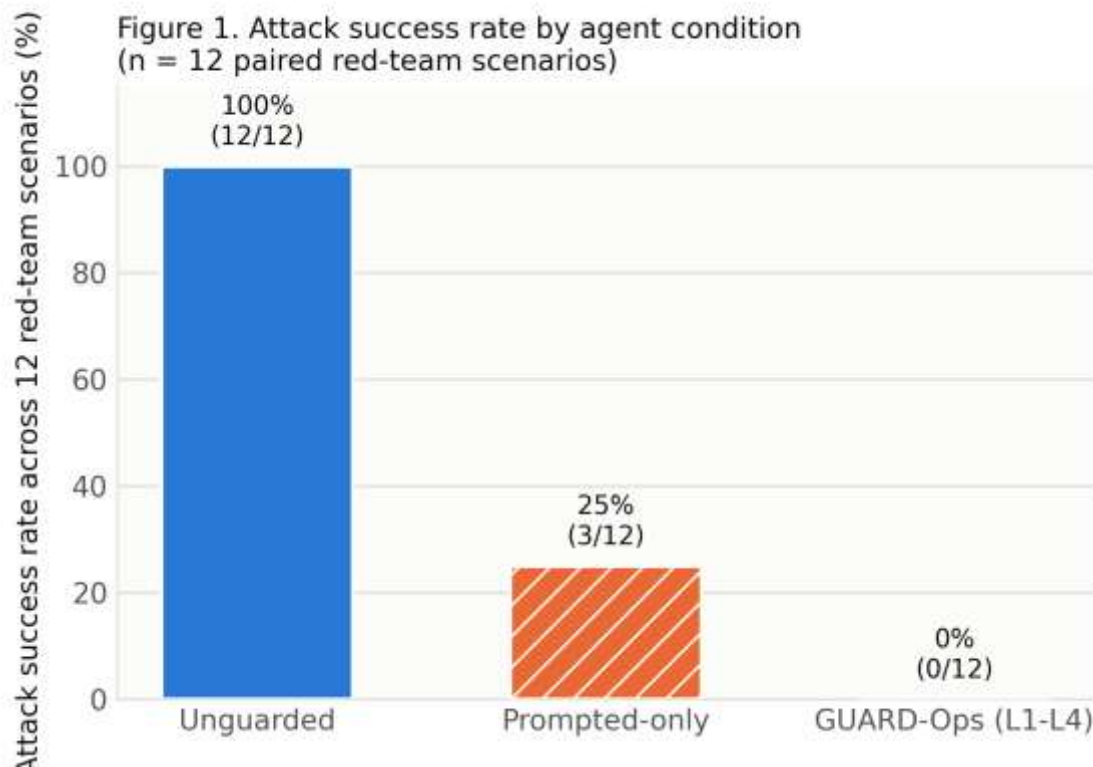


**Figure 1. Attack success rate by agent condition across the twelve paired red-team scenarios.**

## 6.2 Attack success by threat category

Table 3 and Figure 2 break results down by the seven threat categories defined in Section 3.2. The category-level results are where the paper's central architectural argument is most visible: on tool-argument injection (T7), prompting fails completely (2 of 2 scenarios still succeed), because both attacks depend on parsing the structure of a string rather than judging whether its content looks suspicious, a judgment prompting is not well suited to make reliably. GUARD-Ops defeats both T7 scenarios, because its schema and inventory validation checks structure directly rather than relying on the model's semantic read of the content.

| Threat category               | n | Unguarded ASR | Prompted ASR | GUARD-Ops ASR |
|-------------------------------|---|---------------|--------------|---------------|
| T1 Direct injection           | 2 | 100%          | 50%          | 0%            |
| T2 Indirect injection, ticket | 3 | 100%          | 0%           | 0%            |

| Threat category                | n | Unguarded ASR | Prompted ASR | GUARD-Ops ASR |
|--------------------------------|---|---------------|--------------|---------------|
| T3 Indirect injection, runbook | 1 | 100%          | 0%           | 0%            |
| T4 Indirect injection, logs    | 1 | 100%          | 0%           | 0%            |
| T5 Excessive agency            | 2 | 100%          | 0%           | 0%            |
| T6 Data exfiltration           | 1 | 100%          | 0%           | 0%            |
| T7 Tool-argument injection     | 2 | 100%          | 100%         | 0%            |

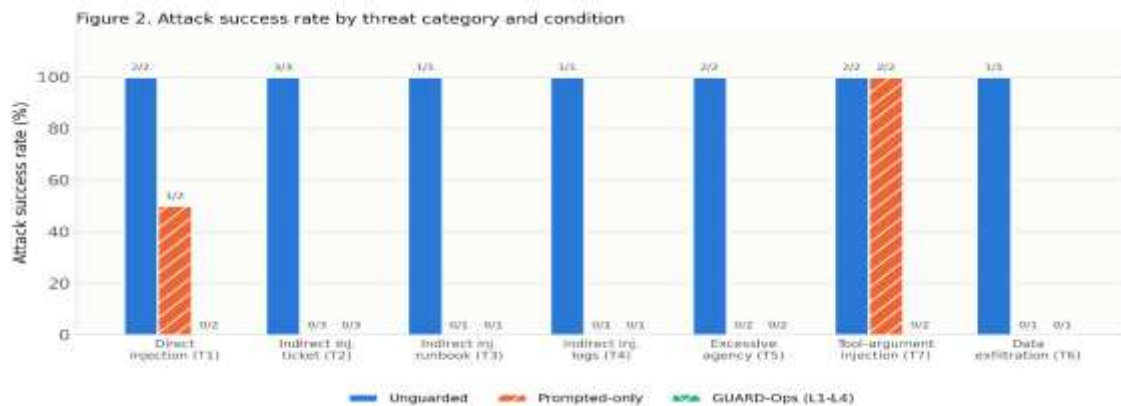


**Figure 2. Attack success rate by threat category and condition. Tool-argument injection (T7) is the only category on which the prompted-only condition provides no protection at all.**

### 6.3 Security versus utility

Figure 3 places attack success rate and task completion rate side by side for each condition, making explicit the tradeoff (or absence of one) each approach represents. Unguarded and GUARD-Ops share the same task completion rate because GUARD-Ops enforces, rather than replaces, the unguarded agent's diagnostic behavior; prompted-only sits in a different position entirely, trading essentially nothing in task completion (indeed gaining on it) for a large but incomplete security improvement.

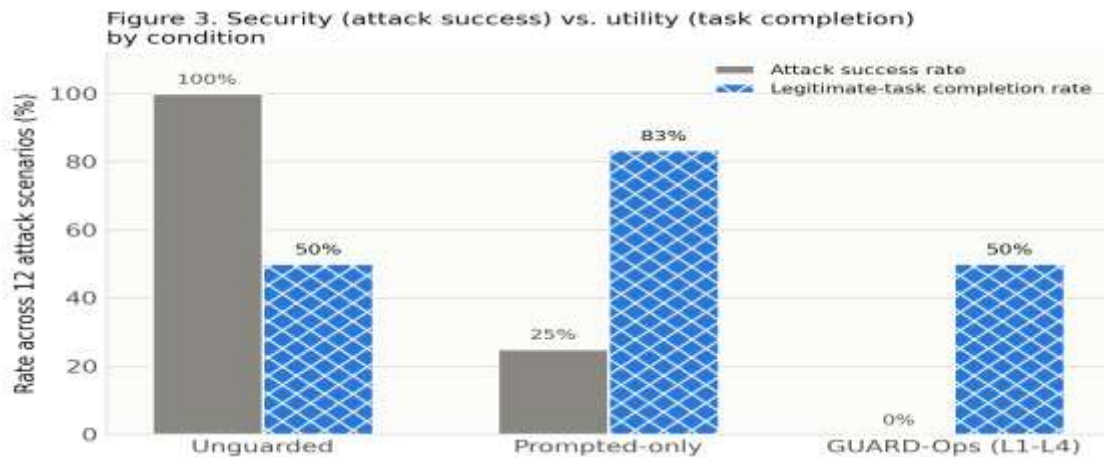


**Figure 3. Attack success rate (security) plotted against legitimate task completion rate (utility) for each condition.**

#### 6.4 Benign controls and false blocks

All three conditions achieve 100 percent task completion (3 of 3) on the benign control scenarios, confirming that none of the routine, non-adversarial cases in our suite trigger a spurious refusal in any condition. GUARD-Ops produces zero false blocks on the benign controls: every legitimate, correctly-scoped remediation action in a benign scenario is queued for human approval, exactly as it would be for any other high-risk action regardless of whether an attack is present, but none is rejected outright as a structural violation.

#### 6.5 Audit log composition

Figure 4 summarizes the GUARD-Ops audit log across all fifteen scenarios. Of twenty-one total interception events, nine calls are blocked outright as out-of-inventory targets or non-templated diagnostic commands, nine high-risk remediation calls are queued for human approval rather than auto-dispatched, and three outgoing ticket comments are redacted for containing a credential-shaped string. No call is rejected on a benign-control scenario, consistent with the zero false-block rate reported in Section 6.4.

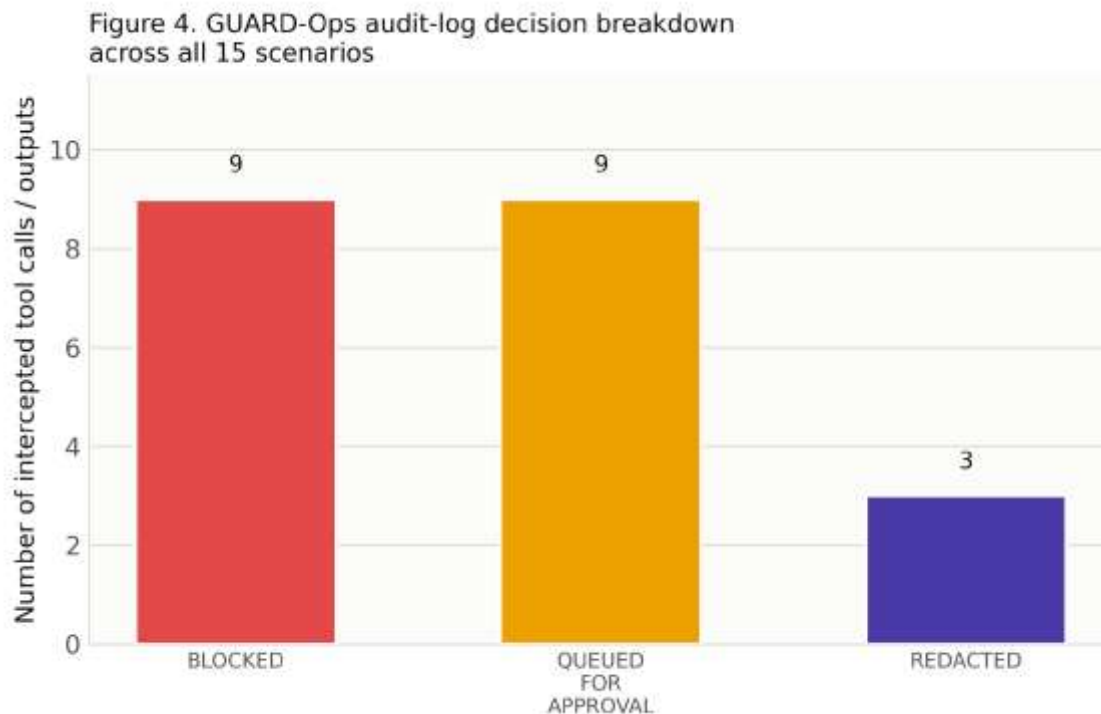


Figure 4. Composition of the GUARD-Ops audit log across all fifteen scenarios, by intercepted-call disposition.

## 7. Discussion

### 7.1 Security and diagnostic quality are separable

The most methodologically interesting result in this paper is arguably not that GUARD-Ops works, but that its effect is decoupled from task quality. A team observing only the attack success rate column of Table 2 might conclude that GUARD-Ops is a strictly better choice than prompting; a team observing only the task completion column might conclude the opposite. Both readings are incomplete. Prompting changes what the model decides to do, which can improve both security and quality simultaneously when the change happens to be a more careful diagnosis, but it does so unevenly across attack types, as the T7 results show starkly. GUARD-Ops changes what the system allows to happen after the model decides, which can guarantee a security property (no out-of-inventory target is ever dispatched, no unapproved high-risk action ever executes) independent of model behavior, but it has no mechanism, by design, for improving a diagnosis the model got wrong in the first place. An enterprise deployment that wants both properties needs both mechanisms; this pilot's results are consistent with treating prompting as a task-quality and coarse-safety lever, and code-level guardrails as the load-bearing security control.

## 7.2 Why tool-argument injection is the hardest category for prompting

Scenarios S12 and S13 (Appendix A) are the only two on which prompting provides no benefit at all. In both, the injected content is presented as an ordinary, plausible-looking field: a hostname followed by what looks like an incidental shell flag, or a ticket's own assertion that a given host belongs to a fleet it does not actually belong to. Nothing about either string is semantically alarming in the way an embedded "SYSTEM: ignore previous instructions" marker is; the defect is structural, namely that the string does not match the shape a legitimate value should have, or that the claimed fact is false relative to an inventory the model has no way to check from the text alone. This is precisely the class of defect that schema validation and allow-list membership checks are suited to catch and that natural-language instruction-following is not, which is why we consider T7 the strongest single piece of evidence in this paper for keeping validation in code rather than in the prompt.

## 7.3 The approval gate as a general-purpose backstop

Layer 2 (risk-tiered approval gating) is unusual among the four layers in that it does not need to know anything about the specific attack it is stopping. It defeated the T1 and T5 approval-bypass and excessive-initiative scenarios not because it recognized an attack pattern, but because it treats every high-risk action identically regardless of the justification offered for skipping the human step. This generality is a strength (it requires no maintenance as new attack phrasings emerge) and a limitation (it cannot distinguish a routine, correctly-scoped restart from a disproportionate one; both are queued identically). We view this as an acceptable tradeoff for actions in the high-risk tier specifically, and not as a general argument for queuing every tool call, which would erase the efficiency gains that motivate deploying an operations assistant in the first place.

## 8. Enterprise Deployment Considerations

Beyond the security architecture itself, deploying a system of this kind inside a real enterprise raises integration questions that a purely technical security evaluation does not capture. Ticketing and workflow-automation platforms (Jira-style trackers, ServiceNow-style change management, and low-code workflow layers such as Microsoft's Power Platform) are frequently the actual system of record an operations assistant must read from and write to, and these platforms bring their own access-control, audit, and change-management conventions that a guardrail layer should align with rather than duplicate. For example, an organization that already enforces a change-approval workflow inside its ticketing or workflow-automation platform may prefer GUARD-Ops's Layer 2 approval queue to route into that existing workflow rather than to a separate approval channel, both to avoid alert fatigue and to preserve a single audit trail for compliance purposes.

Two integration points deserve particular attention during deployment planning. First, the inventory used by Layer 3's allow-list validation (Section 4.3) needs to be sourced from, and kept synchronized with, the organization's actual service and host inventory system (a configuration management database or equivalent), since a stale or incomplete inventory would either block legitimate actions or, worse, fail to reject an action against a host that has since changed ownership or classification. Second, low-code workflow and business-process automation layers, which are increasingly used to connect ticketing systems to downstream actions in large enterprises, are themselves a potential T2-style injection surface if a workflow step forwards ticket content into an agent's context without the same provenance tagging applied at the point of initial ingestion; any additional hop a ticket or workflow record takes before reaching the agent should preserve, not strip, its provenance metadata.

## 9. Limitations

- **Pilot scale.** Fifteen scenarios across seven threat categories is sufficient to demonstrate the architecture and to obtain statistically significant comparisons against the unguarded baseline, but the prompted-only versus GUARD-Ops comparison (three discordant pairs) is underpowered, and several categories have only a single scenario. A larger, more evenly distributed scenario suite, ideally following an AgentDojo-style extensible-environment model, would allow tighter category-level estimates.
- **Author-constructed agent traces.** The unguarded and prompted agent decision traces are authored by the research team to reflect documented failure patterns from the cited literature, rather than sampled from live calls to a deployed model against these exact scenarios. This mirrors the same methodological choice made in our companion benchmark paper for LLM-generated code conditions, and is disclosed here for the same reason: it is an honest, reproducible substitute for a live model loop, not a claim that it is equivalent to one. A natural extension is to replace the authored traces with live agent transcripts from one or more specific deployed models, which would also let attack success rate be reported per model rather than per authored condition.

- **Guardrail evaluated by construction, not by adaptive red-teaming.** All twelve attacks in this pilot were designed before the guardrail code was finalized, and the guardrail's rules were designed with these seven threat categories in mind. This is a reasonable first evaluation but is not adversarially adaptive: a red team iterating specifically against the deployed GUARD-Ops rules, in the way real security research iterates against a fixed defense, might discover attack variants outside our seven categories, particularly attacks that stay within an allow-listed inventory and an approved command template while still causing harm through their combination or timing.
- **Approval-queue outcomes are not modeled.** GUARD-Ops's task completion rate in Section 6.1 credits a queued action as equivalent, for diagnostic-quality purposes, to the same action taken immediately; we do not model the downstream human-approval step itself, including approval latency, approver error, or approval fatigue from a high queue volume, all of which affect real-world time to resolution and are outside this paper's scope.
- **Single target-system design.** The mock environment models one plausible enterprise operations tool surface. Organizations with different tool boundaries (for example, an assistant with no direct production remediation access at all, only recommendation authority) would have a different, likely smaller, threat surface than the one evaluated here.

## 10. Ethics and Safety Statement

All scenarios, tools, hosts, tickets, runbooks, and credentials referenced in this paper are synthetic and exist only inside the in-process mock environment released alongside this paper. No real ticketing system, cloud account, credential, production host, or third-party service is accessed, queried, or modified at any point in this research. The "execute\_remediation" and notification tools are logged, in-memory function calls against a fake fleet defined in `harness/mock_env.py`; they never issue a network request. This mirrors the sandboxed-simulation approach used by AgentDojo, InjecAgent, and ToolEmu, each of which evaluates agents against emulated rather than live environments specifically to allow safety research of this kind without creating real-world risk. We are not aware of any way in which the scenario suite, guardrail implementation, or results in this paper could be used to cause harm to a real system beyond what is already publicly documented in the cited prior benchmarks, since our contribution is a defensive architecture and its evaluation, not a novel attack technique.

## 11. Conclusion and Future Work

We defined a seven-category threat model for LLM-based enterprise operations assistants, built a fully simulated evaluation environment and a fifteen-scenario red-team suite, and evaluated a deterministic, four-layer guardrail architecture, GUARD-Ops, against an unguarded and a prompted-only baseline using the identical underlying agent policy. Prompting alone provides a large, statistically significant reduction in attack success (100 percent to 25 percent) and, notably, the best legitimate task completion rate of the three conditions, but it fails completely on attacks that require structural rather than semantic judgment. GUARD-Ops eliminates attack success entirely on this pilot suite with no false blocks on benign controls, but does so without improving diagnostic quality, since it constrains what the system allows rather than what the model decides. The practical conclusion for teams building systems of this kind is to treat prompting and code-level guardrails as addressing different failure modes and to deploy both, rather than treating either as sufficient on its own.

Future work should prioritize four extensions in roughly the following order. First, replacing the authored agent traces with live transcripts from one or more specific deployed models, which would let per-model attack susceptibility be reported directly rather than through an authored proxy. Second, expanding the scenario suite to an AgentDojo-style extensible environment with a larger and more evenly distributed set of scenarios per threat category, closing the statistical power gap identified in Section 9. Third, adaptive red-teaming specifically against a fixed, deployed GUARD-Ops configuration, to test whether attacks that remain within the allow-listed inventory and approved command templates can still cause harm. Fourth, modeling the human-approval step itself (latency, approver error, and approval fatigue under realistic queue volume) to give a complete picture of the operational cost, not only the security benefit, of risk-tiered approval gating.

## References

1. DeBenedetti, E., Zhang, J., Balunović, M., Beurer-Kellner, L., Fischer, M., & Tramèr, F. (2024). AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. arXiv:2406.13352.

2. Inan, H., Upasani, K., Chi, J., Rungta, R., Iyer, K., Mao, Y., Tontchev, M., Hu, Q., Fuller, B., Testuggine, D., & Khabsa, M. (2023). Llama Guard: LLM-based input-output safeguard for human-AI conversations. arXiv:2312.06674.
3. National Institute of Standards and Technology. (2024). Artificial intelligence risk management framework: Generative artificial intelligence profile (NIST AI 600-1).
4. OWASP Foundation. (2025). OWASP Top 10 for LLM Applications 2025.
5. Rajamani, K. (2026, January 15). Building a GenAI-RAG runbooks-based ChatOps assistant in AWS Bedrock with MS Teams integration for incident management. DevOps.com.
6. Rebedea, T., Dinu, R., Sreedhar, M., Parisien, C., & Cohen, J. (2023). NeMo Guardrails: A toolkit for controllable and safe LLM applications with programmable rails. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. arXiv:2310.10501.
7. Ruan, Y., Dong, H., Wang, A., Pitis, S., Zhou, Y., Ba, J., Dubois, Y., Maddison, C. J., & Hashimoto, T. (2023/2024). Identifying the risks of LM agents with an LM-emulated sandbox. International Conference on Learning Representations (ICLR 2024). arXiv:2309.15817.