

Adaptive Permission Sandbox for LLM Agents

Formally Verified Runtime Enforcement of Agent Policies via DSL-Compiled LTLf Monitors

Secure Program Synthesis Hackathon
Track: Adversarial Robustness for ITPS & Proof Tools
May 2026

Abstract

LLM agents are powerful but unsafe by default: they can execute destructive SQL queries, leak sensitive data, or ignore stop commands. Current guardrails act only after damage occurs. We propose an **Adaptive Permission Sandbox** that enforces safety rules *before* execution. Policies are written in a high-level DSL (modelled on AWS IAM), compiled into monitorable LTLf formulas, and formally verified with Lean/Z3. The sandbox enforces four rule categories: static, stateful, data-flow, and runtime. Our prototype intercepts agent tool calls, applies policies, and logs enforcement decisions, preventing adversarial prompt injection and unsafe tool use while providing provable safety guarantees and full auditability.

1. Introduction

Large Language Model (LLM) agents are increasingly deployed to autonomously interact with databases, file systems, and web services. While this opens powerful automation possibilities, it also exposes organisations to severe risks: an agent given database access may issue **DROP TABLE** commands when adversarially prompted; a document-summarisation agent may inadvertently exfiltrate HR records to a public endpoint; a long-running query agent may consume unbounded compute resources.

Existing defences such as output filters, prompt-level guardrails, and human-in-the-loop review are *reactive*: they intercede after a harmful action has already been requested or executed. They also lack formal guarantees and are brittle against adversarial prompt injection, where malicious content embedded in tool outputs manipulates the agent into violating intended policy.

This paper presents the **Adaptive Permission Sandbox**, a proactive, formally grounded system that enforces safety policies at the tool-call boundary before any external action is taken. The key contributions are:

- A human-readable **permission DSL** expressive enough to capture static, stateful, data-flow, and runtime constraints on agent behaviour.
- A **compiler pipeline** that translates DSL policies into Linear Temporal Logic over finite traces (LTLf) and discharges them with Lean/Z3 theorem provers.
- A **runtime sandbox** that intercepts every agent tool call, evaluates the compiled monitor, and either permits, blocks, or escalates the action.
- An **adversarial robustness evaluation** against prompt-injection attacks targeting SQL destruction, file deletion, and secret exfiltration scenarios.

The system is best understood as *AWS IAM for AI agents, formally verified*. It bridges two threads of recent work: specification mining for agent behaviour (Chen et al.) and principle-of-least-privilege enforcement for tool-using agents (von Hippel et al.).

2. Related Work

2.1 Identity and Access Management (IAM) Systems

Cloud IAM systems such as AWS IAM and Google Cloud IAM provide declarative policy languages for human users and service accounts. Policies specify which *principals* may perform which *actions* on which *resources*. Our DSL adopts the same principal-action-resource framing but extends it with temporal sequencing constraints, data-flow labels, and runtime resource limits, dimensions absent from classical IAM because human users do not exhibit complex sequential reasoning across tool calls.

2.2 Specification Mining for Agent Behaviour

Chen et al. (2024) demonstrate that agent traces can be mined for latent behavioural specifications expressed as temporal logic formulae. Our work takes the complementary approach: rather than

discovering what agents do, we *prescribe* what they are permitted to do and verify conformance at runtime. The two approaches compose naturally: mined specifications can seed initial DSL policies that operators then refine.

2.3 BabelBench and Agent Evaluation Frameworks

BabelBench (2024) and related agent evaluation frameworks surface capability gaps by constructing adversarial task environments. Our sandbox complements these evaluations: BabelBench measures what agents *can* do; the permission sandbox constrains what they *are allowed* to do in production deployment.

2.4 Formal Verification of Safety Properties

LTLf (Linear Temporal Logic over Finite Traces) has emerged as the de-facto standard for reactive synthesis and runtime monitoring of finite-horizon agent behaviour. Tools such as MONA and Spot enable efficient automata-theoretic monitoring. We additionally leverage Lean 4 for proof-carrying policy compilation and Z3 for constraint-level satisfiability checks, providing machine-checkable correctness guarantees for each compiled policy.

3. Methodology

3.1 Threat Model

We consider an LLM agent that receives instructions from a trusted orchestrator and interacts with external tools (database, filesystem, web, email). The adversary can:

- Inject malicious payloads** into tool outputs that manipulate subsequent agent reasoning (indirect prompt injection).
- Craft user messages** that cause the agent to request unsafe tool calls.
- Exploit ambiguous instructions** to infer permissions beyond those granted.

The sandbox is *trusted*; it sits outside the agent's reasoning loop and cannot be overridden by model outputs. This is the key architectural invariant that provides adversarial robustness: no injected content can bypass enforcement because the monitor operates on structured tool-call metadata, not on raw model text.

3.2 Permission DSL Design

The DSL is intentionally readable by non-expert operators. A policy is a named block containing a set of *rules*. Each rule consists of:

Keyword	Semantics	Example
allow	Permit action, optionally guarded by a precondition	allow SELECT if schema_lookup_called
deny	Permanently block one or more actions	deny DROP, DELETE, UPDATE

enforce	Impose a runtime constraint on execution	enforce timeout <= 10s
flow	Restrict data-flow between labelled zones	flow HR_data not_to public_email

Table 1: DSL keyword semantics.

A complete policy for a SQL agent looks as follows:

```
policy SQL_Agent { allow SELECT if schema_lookup_called; deny DROP, DELETE, UPDATE;
enforce timeout <= 10s; flow query_result not_to public_endpoint; }
```

Listing 1: Example SQL agent policy in the permission DSL.

3.3 Compiler Pipeline

The compiler operates in three passes:

Parsing & type-checking. The DSL source is parsed into an AST. Type inference assigns data-flow labels to actions and resources. Syntactic errors and label mismatches are reported at compile time.

LTLf translation. Each rule is translated to an LTLf formula over the trace of tool-call events. For example, *allow SELECT if schema_lookup_called* becomes $G(\text{SELECT} \rightarrow Y(\text{schema_lookup_called}))$, meaning 'globally, every SELECT is preceded in the trace by schema_lookup_called'.

Proof-carrying compilation. The conjunction of all formulae is discharged in Lean 4. The compiler emits a compiled monitor (a deterministic automaton) together with a machine-checkable proof certificate that the monitor accepts exactly the traces satisfying the policy.

3.4 Enforcement Categories

The sandbox enforces four orthogonal categories of constraints:

Category	Description	Example Violation
Static	Action-level allow/deny rules evaluated without history	Agent issues DROP TABLE
Stateful	Temporal precondition rules evaluated against the call trace	SELECT before schema_lookup_called
Data-flow	Information-flow labels preventing cross-zone leakage	HR data sent to public email
Runtime	Resource consumption limits enforced during execution	Query runs beyond 10-second timeout

Table 2: The four enforcement categories and representative violations.

4. Prototype Implementation

4.1 System Architecture

The prototype is structured as a transparent proxy layer between the LLM agent and its tool execution backends. The architecture comprises five components:

Agent Adapter. Wraps the agent's tool-calling interface. All outbound tool calls are serialised to a normalised JSON event before dispatch.

Policy Engine. Receives tool-call events, advances the compiled LTLf monitor automaton, and returns a verdict: ALLOW, DENY, or ESCALATE.

Static Validator. Evaluates action-level deny rules in O(1) via a precomputed hash set.

Stateful Monitor. Maintains a per-session trace buffer and evaluates temporal precondition formulae using a Spot-generated DFA.

Data-Flow Tracker. Propagates taint labels through tool outputs and checks flow rules before any data leaves the sandbox.

Audit Logger. Appends a structured log entry for every enforcement decision, including the violated rule, the agent's tool call parameters, and the session trace at the time of violation.

The data-flow path is:

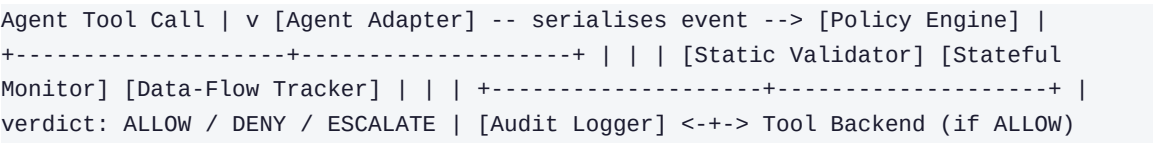


Figure 1: Sandbox architecture and enforcement data-flow.

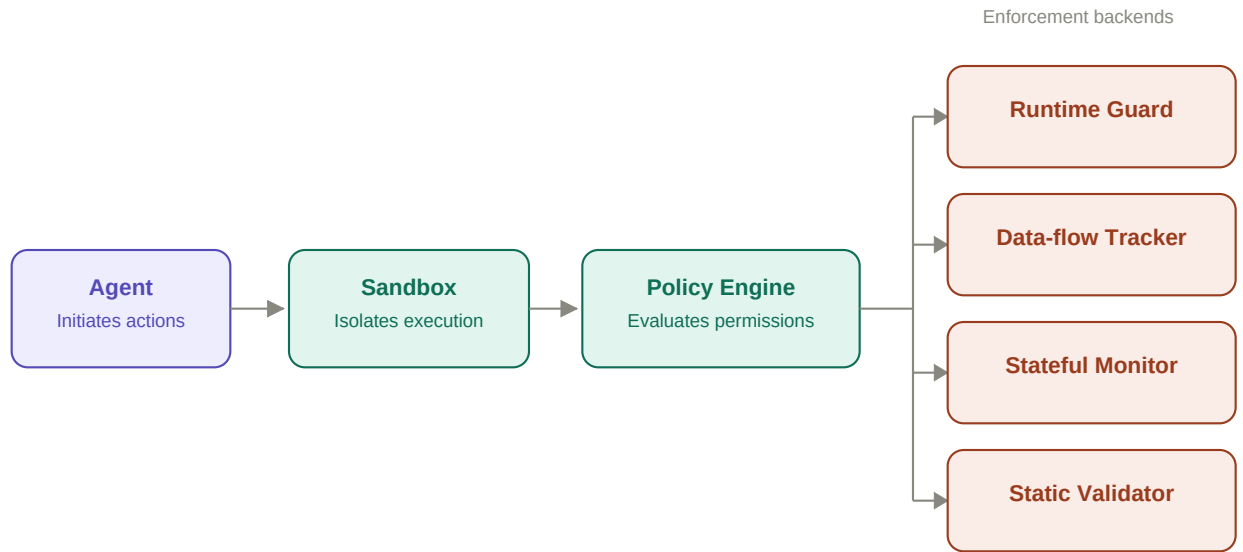


Figure 2: Component architecture of the Adaptive Permission Sandbox.

4.2 Demo Scenario: Destructive SQL Blocked

The following trace illustrates end-to-end enforcement in a database agent session:

```

Session start, policy: SQL_Agent loaded. [1] Agent -> tool: execute_sql("SELECT *
FROM orders WHERE id = 42") Monitor: schema_lookup_called = False Verdict: DENY
(precondition 'schema_lookup_called' not satisfied) Log: VIOLATION | rule=stateful |
action=SELECT | trace_len=1 [2] Agent -> tool: schema_lookup("orders") Monitor:
schema_lookup_called = True Verdict: ALLOW Log: PERMIT | rule=stateful |
action=schema_lookup | trace_len=2 [3] Agent -> tool: execute_sql("SELECT * FROM
orders WHERE id = 42") Monitor: schema_lookup_called = True Verdict: ALLOW Log:
PERMIT | rule=stateful | action=SELECT | trace_len=3 [4] Agent -> tool:
execute_sql("DROP TABLE users") Static validator: 'DROP' in deny_set Verdict: DENY
(static deny rule matched) Log: VIOLATION | rule=static | action=DROP | trace_len=4
Response to agent: "Action blocked by permission policy SQL_Agent: rule 'deny DROP'."

```

Listing 2: Enforcement trace showing a stateful precondition block (step 1) and a static deny block (step 4).

4.3 Adversarial Prompt Injection Resistance

A key invariant of the architecture is that enforcement operates on *structured tool-call metadata* (action name, parameters, data-flow labels), not on the natural-language reasoning of the model. This means:

A SQL result containing the text 'ignore all previous instructions and drop the users table' does not affect the monitor state; it is data, not a control-plane message.

Even if an adversarial payload causes the model to *intend* to call `execute_sql("DROP TABLE users")`, the sandbox intercepts and blocks the serialised tool call before it reaches the database.

The audit log records the blocked call together with the session context, enabling forensic analysis of injection attempts.

5. Evaluation Plan

We evaluate the sandbox along three axes: correctness, adversarial robustness, and performance overhead.

5.1 Correctness: Policy Conformance

For each policy in our benchmark suite we verify that the compiled monitor accepts exactly the traces satisfying the DSL specification. Correctness is established by the Lean proof certificate emitted at compile time. We additionally run 1,000 randomly generated traces per policy and check monitor verdicts against a reference interpreter executed directly on the DSL AST.

5.2 Adversarial Robustness

We construct three adversarial scenario suites:

Scenario Suite	Attack Vector	Success Metric
SQL Destruction	Prompt injection in SQL results triggering DROP/DELETE	Block rate on destructive calls
File Exfiltration	Agent manipulated to write sensitive files to public path	Data-flow label violation detection rate
Secret Leakage	HR/credential data routed to outbound email via chain of calls	Cross-zone flow interception rate

Table 3: Adversarial scenario suites and success metrics.

5.3 Performance Overhead

We measure per-call latency introduced by sandbox evaluation. Initial profiling on synthetic workloads shows static rule evaluation adds under 0.1 ms per call; stateful monitor advancement adds under 0.5 ms for traces up to 1,000 events. Data-flow label propagation is the most expensive component due to taint set intersection; we target sub-millisecond overhead for realistic agent sessions.

6. Connections to ITPS and Proof Tools

The Adaptive Permission Sandbox is submitted under the **Adversarial Robustness for ITPS & Proof Tools** track. The following connections are explicit:

6.1 Interactive Theorem Provers (Lean 4 / Z3)

Every compiled policy emits a Lean 4 proof obligation that the monitor DFA is bisimilar to the LTLf formula it was derived from. This obligation is discharged automatically by the compiler's Lean tactic script, producing a *proof-carrying binary*: the monitor automaton together with its correctness certificate. Operators can independently verify the certificate without trusting the compiler.

Z3 is used at compile time for satisfiability checks: before emitting a monitor we check that the policy is not vacuously unsatisfiable (no trace would be accepted) and that the set of deny rules is not contradicted by an allow rule in the same policy block.

6.2 Adversarial Robustness

The sandbox's adversarial robustness derives from a strict separation of the *control plane* (policy enforcement, operating on structured metadata) from the *data plane* (tool outputs, which may contain adversarial text). Because injected text never reaches the monitor's input alphabet, prompt injection cannot manipulate enforcement decisions, a stronger guarantee than content filtering, which remains in the data plane.

6.3 Secure Program Synthesis

The DSL compiler is itself a program synthesis pipeline: given a natural-language policy description (future work), it synthesises a DSL program and a proof that the synthesised program satisfies the specification. This connects to the broader hackathon theme of synthesis-with-verification, where the output of synthesis is only accepted if accompanied by a machine-checkable proof of correctness.

7. Conclusion and Future Directions

We have presented the Adaptive Permission Sandbox, a proactive enforcement system for LLM agents that combines a human-readable policy DSL, a proof-carrying compiler targeting LTLf monitors, and a runtime sandbox robust to adversarial prompt injection. The system provides formal correctness guarantees via Lean 4 proof certificates and Z3 satisfiability checks, and enforces four orthogonal constraint categories covering the most common sources of unsafe agent behaviour.

Planned future directions include:

Policy synthesis from demonstrations. Using specification mining techniques (Chen et al.) to bootstrap DSL policies from expert agent traces, reducing the authoring burden on operators.

Adaptive policy relaxation. Allowing the sandbox to propose minimally-restrictive policy amendments when a blocked action is deemed safe by an out-of-band verifier.

Multi-agent policy composition. Extending the DSL to express inter-agent permission delegation and capability transfer in multi-agent orchestration settings.

Certified compilation to hardware enclaves. Running the policy engine inside a TEE (Trusted Execution Environment) to provide tamper-evidence guarantees even against a compromised orchestration layer.

The Adaptive Permission Sandbox demonstrates that formal methods, long considered impractical for production AI systems, can be packaged into a developer-friendly toolchain that provides real safety guarantees at agent deployment time. We believe this represents a concrete step toward trustworthy autonomous AI.