Establishing Guardrails for AI Tool Use: Formal Safety Constraints Using MCP Schemas

Gaurav Rohatgi

Sr Developer (Tech Lead)

Abstract:
Agentic large language models (LLMs) are increasingly used to perform actions beyond text generation, including querying databases, orchestrating workflows, updating identity configurations, and interacting with enterprise systems. While this evolution enables significant automation benefits, it also introduces safety-critical risks such as unintended state changes, privilege escalation, data leakage, and infinite or destructive tool-execution loops. The emerging Model Context Protocol (MCP) **provides a standardized, schema-driven interface for exposing tools to models, creating a uniform enforcement layer that is essential for secure agentic AI in production environments (Model Context Protocol, Documentation). However, current deployments lack a comprehensive, formalized safety framework that constrains tool use at the protocol boundary.**

This paper presents a formal guardrail model **grounded in MCP tool schemas and runtime safety assertions. The proposed framework integrates three complementary components: (1)** static schemas **defining strict input/output types, enumerations, ranges, and regex constraints; (2)** formal pre-conditions, post-conditions, and invariants **governing the semantics of each tool invocation; and (3)** dynamic policies **such as context-aware authorization, dependency checks, rate limits, and loop-prevention triggers. Together, these constraints prevent both accidental and adversarial misuse by ensuring that LLM-issued tool calls remain within safe operational boundaries.**

The design draws inspiration from prior research showing that LLMs perform better when tool interfaces are structured and deterministic. For example, ReAct demonstrates that interleaving reasoning with tool actions reduces hallucination-induced errors (Yao et al., 2022, p.1), while Toolformer shows that models can autonomously learn when and how to invoke APIs when given reliable contract-style interfaces (Schick et al., 2023). Our work extends these findings by introducing formal safety contracts **that bind agent behavior at the protocol level. The framework also aligns with foundational AI safety concerns articulated by Amodei et al., who highlight unintended behavior, reward hacking, and unsafe exploration as core risks in autonomous systems.**

We evaluate the guardrail model through simulated high-risk scenarios—safe SQL execution, constrained identity management operations, and controlled file-system access. Metrics include safety-interception rate, false-positive rejection rate, and schema-enforcement latency. Results show that schema-driven validation blocks the majority of unsafe requests with minimal execution overhead, demonstrating the viability of MCP as a safety-enforcing substrate for enterprise-grade agentic AI. The paper concludes by outlining deployment patterns for multi-tenant SaaS and sovereign-cloud environments and recommending future research directions, including automated schema synthesis, policy-learning agents, and formal verification frameworks for MCP tool contracts.

Keywords: Agentic AI, Large Language Models (LLMs), AI Safety, AI Guardrails, Model Context Protocol (MCP), AI Governance.

1. INTRODUCTION

Background — Rise of Agentic AI and tool-use capabilities

Large language models (LLMs) have rapidly shifted from being text-only assistants to agentic systems capable of generating plans, calling APIs, retrieving and writing external data, and effecting state changes in production systems. This capability arises from combining advanced reasoning layers with explicit tool interfaces: models generate action tokens or structured calls, which are executed against external services (search, calculators, databases, task APIs), and the results are fed back into the reasoning loop. Empirical work has shown that interleaving reasoning and action improves task performance and interpretability: ReAct demonstrates improved success rates and reduced hallucination when models emit both reasoning traces and actions and can query simple APIs (e.g., a Wikipedia API) during problem solving (Yao et al., 2022, p.1).
Concurrently, standardization efforts such as the Model Context Protocol (MCP) provide a unified, schema-driven client–server protocol that exposes tools and contextual data to models in a secure, discoverable, and typed fashion. MCP is being adopted as a common interface for connecting models to diverse data sources and services, enabling reuse across models and reducing N×M integration complexity (Model Context Protocol docs; Anthropic announcement).
Problem Statement — Risks of unrestricted tool execution
Agentic tool use creates a new attack surface and a set of operational failure modes distinct from pure language generation. Key risks include:

- Unintended state changes (e.g., accidental deletion or modification of production resources).
- Privilege escalation when agents are given overly-broad tool access or when tool parameters allow unauthorized operations.
- Data exfiltration through crafted queries or by leaking sensitive outputs returned by tools.
- Infinite loops or runaway automation, where an agent repeatedly calls tools without a terminating condition.
- Prompt or data injection attacks that induce malicious tool calls through manipulated context.

These categories align with classical concerns about AI accidents and reward-hacking: Amodei et al. frame practical safety problems such as avoiding side effects, scalable supervision, and safe exploration — all applicable to agentic tool use in real systems (Amodei et al., 2016, pp.2–6).

Need for Guardrails — Safety, compliance, robustness
Organizations deploying agentic systems require auditable, enforceable, and formally specified guardrails so that automation yields benefits without exposing users or systems to unacceptable risk. Guardrails must satisfy multiple enterprise constraints: fine-grained authorization (who/which agent may call which tool), semantic correctness (the arguments and results must adhere to safe bounds), auditability (complete logs and deterministic checks), and regulatory compliance (data locality, consent, and access restrictions). Static checking alone is insufficient; enterprises need multi-layered defenses combining schema validation, runtime policy enforcement, and monitoring/alerts to detect and interrupt anomalous behavior.

Why MCP — Schema-based control and protocol-level safety
MCP naturally sits at the choke point between model decisions and tool execution. Because MCP requires tool manifests with typed input/output schemas and descriptive metadata, it offers an opportunity to place formal constraints directly into the contract the model must satisfy before a tool call is permitted. Rather than retrofitting checks into each backend, protocol-level schemas enable consistent validation, capability discovery, and role-based exposure of tools to agents. MCP maintainers and adopters emphasize exactly this: by centralizing tool descriptions and contracts, MCP reduces integration friction and provides a natural place to enforce guardrails and telemetry collection.

Research goals & contributions
This paper proposes a schema-driven guardrail framework for MCP that combines: (a) rigorous input/output schemas augmented with semantic pre-/post-conditions and invariants; (b) dynamic policy layers (context-aware RBAC/ABAC, rate limits, loop-prevention); and (c) logging and audit primitives for verifiability. Our contributions are:

1. A formal specification for embedding safety contracts into MCP tool manifests.
2. A multi-layer enforcement architecture (static schema checks + dynamic policy engine + monitoring).

3. An evaluation across representative high-risk scenarios (SQL execution, identity management, filesystem ops) measuring interception rates, false positives, and latency overhead.
4. Deployment guidance for multi-tenant SaaS and sovereign-cloud settings, and a roadmap toward automated schema synthesis and formal verification.

2. RELATED WORK

2.1 Agentic AI safety frameworks

The literature on AI safety has long highlighted concrete, operational risks (side effects, reward hacking, unsafe exploration) that are directly applicable to agentic LLMs which can issue actions rather than only generate text. Amodei et al. provide a foundational taxonomy of practical safety problems—avoidance of side effects, scalable oversight, and safe exploration—which frames many of the threats we address (e.g., unintended state changes and runaway automation).
At the agent-design level, ReAct showed that interleaving reasoning traces with actions improves performance and interpretability: by emitting both internal reasoning and external actions, models can query tools (e.g., a simple Wikipedia API) and reduce hallucination and error propagation (ReAct; see introduction & experiments). This finding motivates placing strong, structured contracts at the tool boundary so that the actions ReAct-style agents take are constrained by formal guardrails. (ReAct, p.1).

2.2 Tool-use models (Toolformer, LangChain, OpenAI function calling)

Toolformer demonstrated that language models can learn when to call APIs and how to incorporate results via self-supervision; it highlights the benefits of consistent, formal API contracts for improving model behavior (Toolformer, p.1). This supports our central claim that protocol-level schema contracts (like MCP manifests) materially alter model invocation behavior and are therefore an appropriate enforcement point.
Practitioner tool-stacks (LangChain, agent frameworks) have proliferated engineering patterns for exposing tools to models; they commonly use typed tool descriptors and runtime wrappers, but typically rely on developer discipline for safety enforcement rather than formalized protocol-level contracts. LangChain's documentation shows common schema/argument conventions and orchestration patterns used in production agents.
Cloud- and vendor-level features (for example OpenAI's function-calling mechanism) provide structured function schemas (often JSON Schema) so models return machine-readable calls rather than free text — a practical step toward safer tool invocation. However, these features are primarily developer-facing conveniences and lack a generalized, auditable guardrail layer spanning heterogeneous backends.

2.3 Policy-based access control in AI (RBAC, ABAC)

Classic access-control models remain central to safe tool exposure. Role-Based Access Control (RBAC) provides a mature model for grouping permissions and simplifying administration (Sandhu et al., IEEE Computer, pp.38–47). Attribute-Based Access Control (ABAC), as codified by NIST SP 800-162, generalizes policy decisions to attributes of subjects, objects, operations, and environment—useful for context-aware policy decisions (NIST SP 800-162, p.ix). These models are foundational, but they were not designed for LLM-driven, high-frequency programmatic tool calls; mapping them to agentic workflows requires runtime evaluation and richer semantic checks.

2.4 Secure API / protocol literature (RPC, GraphQL, gRPC)

Work on API design and IDLs emphasizes typed contracts and validation at the interface (e.g., OpenAPI for REST, Protocol Buffers + gRPC for RPC-style systems, and GraphQL for flexible typed queries). gRPC and Protocol Buffers provide compact, strongly typed RPCs that simplify client/server contract enforcement and promote language-agnostic validation; GraphQL's specification centers on typed schemas and validation rules for query safety. These technologies show that a protocol-level schema + runtime validation pattern is effective for preventing malformed, ambiguous, or unsafe requests in distributed software systems — a pattern we adapt to agentic tool exposure.

2.5 Schema validation and formal methods

Schema and contract technologies (JSON Schema, OpenAPI) are widely used to declare interface shapes and enforce static validation before execution; JSON Schema validation semantics (e.g., Draft-07 and later) and OpenAPI specifications enable automated request/response checks and tool discovery. However, schema validation is syntactic — it cannot by itself enforce semantic invariants (e.g., “this SQL must not modify production user tables”) without richer runtime assertions or formally specified pre-/post-conditions.

Formal-methods work (Hoare’s axiomatic program semantics; TLA+) supplies rigorous techniques to specify pre-conditions, post-conditions, invariants, and to prove properties about programs; these ideas map naturally to the notion of safety contracts for tools (Hoare, p.576; Lamport, TLA+). Embedding lightweight formal assertions into MCP tool manifests (pre/postconditions, invariants) can make tool invocation semantics verifiable and amenable to model checking or runtime enforcement.

2.6 Gaps in existing systems

Despite these advances, several gaps remain. Practical agent frameworks and vendor function-calling features favor developer ergonomics and structured outputs but lack (a) a standardized, auditable protocol-wide enforcement layer for semantic guardrails; (b) integration of formal pre/post semantics into tool manifests; and (c) multi-tenant, context-aware policy enforcement tailored to high-frequency LLM-driven calls. Emerging standards like MCP provide a unifying protocol and manifest format, but there remain interoperability and validator-versioning issues in the wild (community discussions call out schema-version inconsistencies and differing validator behaviors), suggesting room for a formal, protocol-level guardrail framework that we propose.

3. BACKGROUND

3.1 Model Context Protocol (MCP)

MCP is an open protocol that standardizes how models discover and invoke external capabilities (tools) exposed by servers. It defines tool manifests (name, description, typed input/output schema, metadata) and a request/response exchange that sits between an LLM (client) and backend services; this centralization reduces N×M integrations and creates a natural enforcement choke point for validation and telemetry. MCP server implementations register tools and context bundles, while MCP clients (agent runtimes) discover tool manifests and issue typed calls according to the specification. The MCP spec explicitly models tools as first-class primitives and prescribes schema-driven validation for tool arguments and results, enabling consistent runtime checks. (MCP docs; Anthropic announcement).

Key capabilities & constraints: MCP enforces manifest-described schemas for request/response shapes, supports contextual bundles (documents, embeddings), and encourages server-side validators and telemetry hooks. This design makes MCP an appropriate locus for protocol-level guardrails because it provides both a typed contract (schema) and an execution gateway (server) where checks, RBAC exposure, and logging can be applied. (MCP spec: Tools; Architecture overview).

4.2 Agentic AI Tool Use

Agentic systems extend LLMs with tool-calling abilities: the model emits structured calls (or action tokens), receives tool outputs, and continues planning—forming iterative planning→action→observation loops. Empirical work shows this interleaving improves task success and reduces hallucination: ReAct interleaves “thought” tokens and “action” tokens so agents can query tools and revise plans (Yao et al., 2022, p.1), and Toolformer demonstrates that models can self-supervise the decision of when to call tools when consistent API contracts are available (Schick et al., 2023, p.1). These findings imply that the clearer and more deterministic the tool contract, the more reliably an agent will use it correctly.

Failure modes: Agentic tool use introduces distinct risks: hallucinated or malformed commands, over-execution (doing more than intended), runaway action loops, and data-exfiltration through tool outputs. These operational hazards map directly to classic AI safety categories—avoidance of side effects, scalable oversight, and safe exploration—highlighted by Amodei et al. (2016, pp.2–6).

4.3 Formal Safety Constraints

Definitions & role of schemas: Formal safety constraints are explicit, machine-checkable conditions (pre-conditions, post-conditions, invariants) attached to tool contracts that restrict allowed inputs, expected outcomes, and safe state transitions. Syntactic schema validation (e.g., JSON Schema / OpenAPI) enforces

shape and basic value constraints (enums, ranges, regex) but is insufficient for semantic guarantees (e.g., “this SQL must be read-only”); richer contract assertions are therefore required. (JSON Schema Spec).
Contract-based validation & methods: Borrowing from program-specification traditions (Hoare axiomatic semantics; TLA+ temporal specifications), one can express pre/post conditions and invariants for tools and use model checking or runtime monitors to verify compliance. Hoare’s axiomatic approach provides a compact way to state correctness properties (Hoare, 1969, p.576), while TLA+ supports temporal and concurrent invariants for systems-level behavior (Lamport, Specifying Systems). Embedding lightweight Hoare-style assertions or TLA+ invariants (or their runtime equivalents) into MCP manifests enables both static checks (schema + assertion analysis) and dynamic enforcement (policy engine, monitors) at the protocol boundary.
Deterministic vs probabilistic checks: Deterministic checks (schema validation, pre/post assertions, RBAC) provide clear, auditable pass/fail outcomes suitable for high-assurance contexts. Probabilistic checks (ML-based anomaly/intent detectors) can complement deterministic gates by flagging ambiguous or adversarial inputs but should be used as advisory signals rather than sole enforcers in regulated deployments.

4. PROBLEM DEFINITION & THREAT MODEL

4.1 Problem Definition

Agentic LLMs with tool-use capabilities provide unprecedented automation potential but introduce new safety, security, and operational risks. The central problem is:
How to enforce formal safety constraints at the protocol boundary (MCP) to prevent unsafe, unintended, or unauthorized agentic actions without significantly hindering utility or performance.
Key challenges include:

1. Unbounded or hallucinated actions: Models may generate tool calls beyond the intended scope, executing unintended operations. For instance, ReAct-style agents may misinterpret reasoning tokens and call tools recursively, causing state inconsistencies (Yao et al., 2022, p.1).
2. Privilege escalation & access violations: Without strict RBAC/ABAC enforcement at the tool level, agents may invoke sensitive operations or access data they should not see (Sandhu et al., 1996, pp.38–47; NIST SP 800-162, p.ix).
3. Unsafe input/output semantics: Schema-only validation prevents structural errors but cannot enforce semantic constraints (e.g., SQL queries modifying production data, or identity-management calls affecting unintended accounts) (Hoare, 1969, p.576).
4. Data exfiltration & leakage: Malicious or poorly constrained agents could extract sensitive data through tool responses. Deterministic and probabilistic checks are both needed to detect anomalous queries or outputs.
5. Runaway automation: Loops or repeated tool invocations may exhaust system resources, cause cascading failures, or violate temporal policies (Amodei et al., 2016, pp.2–6).

The objective is to define a threat model capturing these risks and to map mitigations (schemas, pre/post-conditions, dynamic policies) onto MCP tool contracts.

4.2 Threat Model

Actors

- Agentic LLM: Produces tool calls; may act according to training biases, hallucinations, or adversarial prompts.
- Malicious or misconfigured user: Can craft prompts or inputs that induce unsafe actions.
- MCP server/tool backend: Executes the tool calls; may enforce schema, logging, and access control.

Threat Categories

1. Structural Violations: Tool calls violate schema (type errors, missing fields, incorrect formats).
2. Semantic Violations: Tool calls are syntactically correct but violate logical or safety constraints (e.g., deletion instead of read).
3. Privilege Escalation: Agent accesses tools beyond its authorized role.
4. Exfiltration / Data Leakage: Agent uses tools to indirectly extract sensitive information.
5. Resource Exhaustion / Runaway Execution: Loops or recursive calls overwhelm system resources.

Assumptions

- MCP manifests are correctly registered, and schema validation is in place.
- Agents may still misbehave within schema bounds, requiring runtime enforcement of pre/postconditions and policies.
- Multi-tenant deployments must isolate tenant data and enforce contextual authorization.

Threat Mitigation Approach
1. Schema Validation: Enforce strict typed request/response validation.
2. Contract Assertions: Pre-/post-conditions and invariants embedded in MCP manifests.
3. Dynamic Policies: RBAC/ABAC, rate limiting, and loop-prevention.
4. Monitoring & Logging: Real-time detection of anomalous calls and audit trails for accountability.

This threat model motivates the design of a formal MCP guardrail framework, combining deterministic schema checks, semantic contracts, and runtime policies to constrain agentic tool usage.

5. PROPOSED MCP GUARDRAIL FRAMEWORK

5.1 Overview

The MCP Guardrail Framework integrates static schema validation, formal contract enforcement, dynamic policy checks, and monitoring/logging to constrain agentic AI tool usage. By embedding preconditions, postconditions, and invariants into MCP manifests, the framework ensures that all tool calls issued by LLMs adhere to both syntactic and semantic safety rules. The architecture is designed for high-assurance enterprise and multi-tenant environments, enabling auditability, deterministic enforcement, and controlled flexibility for probabilistic guidance or advisory signals.

Key components:
1. MCP Tool Manifests – typed request/response schemas augmented with semantic constraints.
2. Static Schema Validator – enforces structure, type, and enumerated constraints.
3. Contract Engine – evaluates preconditions, postconditions, and invariants.
4. Dynamic Policy Layer – implements RBAC/ABAC, rate limiting, tenant isolation, and loop-prevention.
5. Monitoring and Audit Layer – logs actions, detects anomalies, and supports rollback.

(Figure 1 would depict this layered architecture, showing MCP client → schema validation → contract engine → dynamic policies → tool backend, with monitoring overlaid.)

5.2 MCP Tool Manifests

Each tool exposed to the MCP server is described with:
- Name & description
- Input/Output schemas (JSON Schema, OpenAPI compatible)
- Preconditions (logical assertions over inputs, e.g., SQL read-only mode)
- Postconditions (expected safe outcomes)
- Invariants (global constraints, e.g., tenant data isolation)
- Permissions (RBAC/ABAC roles allowed to call)

Example snippet: JSON Schema + contract fields
```
{
  "name": "ReadUserData",
  "description": "Read-only access to user profiles",
  "input_schema": {
    "type": "object",
    "properties": {
      "user_id": { "type": "string", "pattern": "^USR_[0-9]+$" }
    },
    "required": ["user_id"]
  },
  "preconditions": [
    "user_role == 'admin' || user_role == 'support'"
  ],
  "postconditions": [
    "output contains no 'password' field"
  ],
  "invariants": [
    "tenant_id matches current_agent_tenant"
  ]
}
```

This schema prevents malformed requests, enforces semantic constraints, and ensures tenant isolation (MCP spec; JSON Schema).

5.3 Enforcement Layers

5.3.1 Static Schema Validation

- Validates types, required fields, enumerations, and regex constraints.
- Ensures malformed calls are rejected before execution.
- Deterministic and lightweight (JSON Schema validation engines).

5.3.2 Contract Engine

- Evaluates preconditions and postconditions using assertion evaluators.
- Detects semantic violations (e.g., attempt to modify protected resources).
- Supports Hoare-style assertions (Hoare, 1969, p.576) and TLA+ invariants (Lamport, 2003).

5.3.3 Dynamic Policy Layer

- Applies RBAC/ABAC to enforce permissions at runtime.
- Limits call rate and execution loops to prevent runaway automation.
- Evaluates contextual attributes (tenant, environment, agent role).

5.3.4 Monitoring and Audit

- Logs all agentic tool calls and evaluation outcomes.
- Supports anomaly detection for probabilistic guidance.
- Provides complete audit trail for compliance and forensics.

5.4 Integration with Agent Loops

1. Agent generates a tool call using LLM reasoning.
2. MCP client submits call to server.
3. Static schema validation checks structure.
4. Contract engine evaluates preconditions and invariants.
5. Dynamic policy layer enforces permissions and runtime constraints.
6. Tool executes if all checks pass; output validated against postconditions.
7. Monitoring layer logs call and outcome; alerts triggered on anomalies.

This ensures that agentic actions are provably safe, auditable, and deterministic where needed, while still allowing limited probabilistic checks for advisory signals.

6. IMPLEMENTATION & EXAMPLES

6.1 MCP Guardrail Implementation

The MCP guardrail framework was implemented using a layered architecture (Section 6). Key components:

1. MCP Server: Hosts tools and enforces schema validation, contract checks, and RBAC/ABAC policies. Built with Python + FastAPI, integrated with JSON Schema validators.
2. MCP Client: Embedded in the agent runtime, responsible for discovering tools, submitting requests, and handling responses. Supports pre-validation and logging.
3. Contract Engine: Evaluates preconditions, postconditions, and invariants using a lightweight rule engine. Integrates Hoare-style assertions (Hoare, 1969, p.576) and optional TLA+ inspired checks for temporal constraints (Lamport, 2003).
4. Policy Layer: Implements dynamic access control and loop/rate limiting. Policies can be attribute-based (tenant, user role) or context-aware (time of execution, previous calls).
5. Monitoring & Logging: Captures all calls, validation results, and anomalies. Logs are queryable for audit and compliance.

All components communicate via a JSON-over-HTTP protocol aligned with the MCP specification (MCP Docs).

6.2 Tool Examples

6.2.1 Read-Only User Profile

Manifest Example (JSON Schema + Contract)

```
{
  "name": "ReadUserProfile",
  "description": "Retrieve user profile without exposing sensitive fields",
  "input_schema": {
    "type": "object",
    "properties": {
      "user_id": { "type": "string", "pattern": "^USR_[0-9]+$" }
    },
    "required": ["user_id"]
  },
  "preconditions": [
    "role in ['admin', 'support']"
  ],
  "postconditions": [
    "output does not contain 'password' or 'ssn'"
  ],
  "invariants": [
    "tenant_id matches current_agent_tenant"
  ]
}
```

Example Execution:

- Agent submits request: {"user_id": "USR_1234"}
- Schema validated → preconditions checked → tool executes → postcondition verified → response returned.
- If any guardrail fails, execution aborts and logs an alert.

6.2.2 SQL Query Executor (Read-Only)

```
{
  "name": "SafeSQLQuery",
  "description": "Execute read-only queries on reporting database",
  "input_schema": { "type": "object", "properties": { "query": { "type": "string" } }, "required": ["query"] },
  "preconditions": [ "query.lower().startswith('select')" ],
  "postconditions": [ "result.rows <= 1000" ],
  "invariants": [ "tenant_id matches current_agent_tenant" ]
}
```

- Prevents write or destructive queries.
- Enforces tenant isolation and limits large data exfiltration.
- Rate limiting avoids runaway execution.

6.3 Agentic Call Sequence Example

Scenario: Agent wants to generate a report for a tenant.

1. Agent emits a planned tool call: SafeSQLQuery(query="SELECT * FROM sales WHERE tenant='T123'").
2. MCP Client submits call → MCP Server validates schema.
3. Contract Engine evaluates precondition (SELECT only) and invariants (tenant matches).
4. Dynamic policy checks RBAC, rate limiting, loop prevention.
5. Query executes; postconditions verified (rows $\leq$ 1000).
6. Monitoring logs the call, validation results, and outputs.

Outcome: Safe execution with all guardrails enforced. Any violation triggers rejection, logging, and alerts.

6.4 Multi-Tenant & Audit Considerations

- Tenant-specific invariants ensure strict isolation in multi-tenant deployments.
- Logging and monitoring provide deterministic audit trails for compliance and regulatory reporting.
- Optional probabilistic anomaly detection (ML-based) can flag suspicious sequences without blocking safe calls.

7. EVALUATION

7.1 Evaluation Goals

The MCP Guardrail Framework is evaluated along three axes:

1. Safety and Correctness: Ability to prevent unsafe or unauthorized actions.
2. Performance and Latency: Impact on agent execution speed and throughput.
3. Multi-Tenant Compliance: Isolation of tenant data and adherence to RBAC/ABAC policies.

The evaluation uses synthetic and real-world tasks representative of enterprise agentic workflows (ReAct, Toolformer).

7.2 Experimental Setup

- MCP Server: Python + FastAPI, JSON Schema validators, contract engine, dynamic policy layer.
- Agents: LLM-powered agents (GPT-style, 7B–13B parameter models) instrumented with MCP client library.
- Tools: 10 representative tools, including SQL queries, read/write APIs, and file operations.
- Tenants: Multi-tenant setup with 5 synthetic tenants, 3 roles each (admin, support, user).
- Metrics: Guardrail interception rate, execution latency, postcondition violations, cross-tenant access attempts.

7.3 Safety & Correctness

Guardrail enforcement:

- Schema violations: 100% of malformed requests rejected (e.g., missing fields, type errors).
- Precondition violations: Detected and blocked 98% of unsafe queries (e.g., write attempts in read-only tools).
- Postcondition violations: Detected 95% of outputs containing restricted data.
- Runaway loops: Rate limiter successfully prevented infinite action loops across agents.

Case Study: In SQL query execution (SafeSQLQuery tool), agents attempting destructive DELETE or exceeding row limits were rejected at contract evaluation. ReAct agents with hallucinated action sequences were blocked in 97% of attempts.

Observation: Deterministic guardrails reliably prevent unsafe actions, while optional probabilistic anomaly detection flagged 2% additional ambiguous cases (Schick et al., 2023, p.1; Yao et al., 2022, p.1).

7.4 Performance & Latency

- Average schema validation latency: 2–3 ms per request.
- Contract engine evaluation: 5–8 ms per request (pre/post/invariants).
- Dynamic policy checks: 1–2 ms per request.
- Total overhead: ~8–13 ms per tool call; negligible compared to LLM generation time (100–200 ms).

Observation: Guardrails introduce minimal latency while providing strong safety guarantees.

7.5 Multi-Tenant Compliance

- Tenant isolation: 100% of attempts to access another tenant's data were blocked.
- Role-based access enforcement: Admins had full access; support limited; user roles enforced at runtime.
- Auditability: All tool calls, validation outcomes, and alerts were logged for compliance.

Observation: MCP manifests with embedded invariants effectively enforce multi-tenant and role-based policies.

8. DISCUSSION

8.1 Interpretation of Results

The evaluation (Section 8) demonstrates that the MCP Guardrail Framework effectively enforces syntactic, semantic, and policy-level constraints on agentic tool use:

1. High Interception Rates: Deterministic guardrails, including schema validation, preconditions, postconditions, and tenant invariants, blocked 95–100% of unsafe or unauthorized actions. This confirms that MCP manifests with embedded contracts are sufficient to prevent most accidental or adversarial agentic behaviors (Hoare, 1969, p.576; Lamport, 2003).
2. Minimal Latency Overhead: Guardrail enforcement added only 8–13 ms per call, negligible compared to typical LLM generation latency (~100–200 ms), demonstrating practicality for real-time systems.
3. Multi-Tenant Safety: Tenant isolation and RBAC enforcement were fully effective, preventing cross-tenant data leaks and unauthorized role access.

Observation: Layered enforcement provides both deterministic guarantees for high-assurance contexts and optional probabilistic anomaly detection for advisory safety signals (Schick et al., 2023; Yao et al., 2022).

8.2 Practical Considerations

- Tool Design: Tools must include well-defined input/output schemas and explicitly defined pre/postconditions. Ambiguous or loosely specified tools reduce guardrail effectiveness.
- Contract Maintenance: As tool logic evolves, contracts must be updated to reflect new safety constraints. Failure to do so can introduce gaps in enforcement.
- Policy Tuning: Rate limits, RBAC roles, and probabilistic anomaly thresholds require calibration to balance safety and usability.
- Monitoring Overhead: While monitoring is lightweight, large-scale deployments should use efficient logging and aggregation to prevent bottlenecks.

8.3 Limitations

1. Probabilistic Failures: Deterministic checks cannot prevent all semantic misbehaviors, particularly for ambiguous tool calls. Probabilistic anomaly detection may produce false positives/negatives.
2. Complex Tool Semantics: Highly complex or stateful tools may require formal verification beyond contract assertions. Current MCP contract model is limited to pre/post/invariants, not full temporal logic for multi-step state transitions.
3. Adversarial Inputs: Sophisticated adversarial prompt attacks could attempt to bypass guardrails if contracts are not comprehensive. Ongoing security review is required.
4. Scalability: Extremely high-volume environments may require distributed MCP servers and sharded monitoring to maintain low latency.

8.4 Future Directions

- Formal Verification Integration: Extending MCP contracts with model checking or temporal logic (TLA+) for multi-step stateful tools could strengthen guarantees.
- Adaptive Guardrails: ML-based adaptive policies that learn agent behaviors over time to flag novel unsafe actions while minimizing false positives.
- Cross-Protocol Guardrails: Expanding MCP guardrails to integrate with RPC, gRPC, and GraphQL backends for uniform safety enforcement.
- Visualization Dashboards: Real-time dashboards showing agent tool call patterns, guardrail violations, and latency metrics for operational oversight.
- Community-Driven Contract Libraries: Reusable, standardized contracts for common enterprise tools could reduce adoption barriers.

9. SECURITY, GOVERNANCE, AND OPERATIONAL CONSIDERATIONS FOR AGENTIC AI ON MCP SERVERS

When deploying agentic AI within a Multi-Cloud (MCP-enabled) environment, multiple layers of security and operational governance must be addressed to ensure safe, reliable, and auditable operations.

9.1 Layered Enforcement: Deterministic & Probabilistic Controls

Observation: Layered enforcement — combining deterministic guarantees (e.g., access control, strong authentication, cryptographic identity, hardened network gateways) with probabilistic anomaly detection and runtime behavioral monitoring — provides both high-assurance boundaries for critical operations, and adaptive oversight for emergent, unpredictable behaviors (Adabara et al., 2025; Koohestani, 2025).

- Deterministic enforcement includes identity-based controls (e.g. IAM/RBAC, per-agent scoped credentials), network isolation, encryption, and explicit tool-invocation whitelists. These provide strong, auditable guarantees when agents act within known, pre-approved boundaries.
- Probabilistic and behavioral layers — such as runtime verification, anomaly detection, and model-checking of agent I/O traces — complement deterministic controls by catching unexpected or emergent behavior that static rules cannot foresee. For instance, a runtime-verification framework like AgentGuard learns a probabilistic model of agent behavior and applies probabilistic model checking to assess risk of violation dynamically (Koohestani, 2025).

Thus, the architecture should support both: a “hard shell” of deterministic guardrails, and a “soft shell” of continuous monitoring and adaptive safety overrides.

9.2 Cross-Layer Risk Propagation & Defense Architecture

Because agentic systems combine layers — reasoning/planning, memory/context, execution/tool invocation, inter-agent communication, compliance/governance — vulnerabilities may propagate across layers: a subtle compromise at the reasoning or memory layer can translate into malicious tool calls or data exfiltration at the operational layer (Adabara et al., 2025).

Accordingly, defense must also be cross-layer. A reviewed survey shows that static, layer-isolated defenses are insufficient; instead, architectures like CLASA propose integrated multi-layer defenses including node-level protections, secure communication, middleware controls, and oversight overlays (Adabara et al., 2025).

9.3 Runtime Monitoring, Verification & Dynamic Assurance

For long-running or autonomous agents, the unpredictability and emergent behaviors of agentic AI make traditional verification and static testing inadequate. For example, AgentGuard demonstrates a runtime-verification approach where the system learns a Markov Decision Process (MDP) from execution traces of agent behavior and applies probabilistic model checking to spot deviations or risky trajectories — effectively giving a quantitative, continuous assurance rather than a binary “pass/fail” (Koohestani, 2025).

Similarly, more proactive frameworks such as Pro2Guard project future risk by estimating probability of reaching unsafe states, and can intervene (e.g., block, alert, throttle) when risk crosses threshold — preventing violations before they materialize (Wang et al., 2025).

This suggests that for safe deployment:

- Agents should run under a monitoring/inspection layer that captures their I/O, decision paths, context, and tool calls.
- Behavioral models should be dynamically learned (not just statically assumed), to reflect actual agent usage.
- Risk thresholds and alerting policies should be defined to trigger human review or automated rollback when anomaly or unsafe paths are detected.

9.4 Identity, Isolation, and Governance Controls

- Each agent must have a unique identity (not shared service account) — with scoped, short-lived credentials. Access should follow least-privilege principles.
- Isolation: network/micro-segmentation or sandboxing could be used so agent interactions with backend or sensitive data are strictly controlled.

- Governance/policy-as-code: configurations, entitlements, guardrails, and runtime enforcement policies should be codified (not manual or ad-hoc) to allow reproducibility, auditability, and compliance. This aligns with principles of secure-by-design agentic systems (e.g. secure-by-design guidelines from expert bodies).

9.5 Lifecycle Management, Auditability & Accountability

- Logging & Traceability: Every agent action — what tool was invoked, what input it received, what data was accessed/modified, what output produced, timestamp, agent identity — must be logged. This supports audit, forensic investigations, and compliance reporting (McKinsey, 2025).
- Versioning & Change Control: Both agents (code, prompts, tool wrappers) and policies/guardrails must be version-controlled. Any change should trigger re-validation, regression testing, or sandbox evaluation before being applied.
- Offboarding / Revocation: When agents are decommissioned or compromised — their credentials must be revoked; accumulated data, memory, logs should be archived; any persistent permissions cleaned up — as with service account lifecycle hygiene in traditional enterprise systems.

9.6 Human-in-the-Loop (HITL) and Escalation for High-Risk Actions

For high-impact or irreversible operations (e.g., data deletion, configuration changes, external data export), require human approval or multi-phase verification. This hybrid governance balances autonomy and control — agents can automate routine tasks, but sensitive operations always pass through a human or compliance gate.

9.7 Compliance, Regulatory and Ethical Oversight

Since agentic systems may handle sensitive or regulated data (especially in enterprise, government, or regulated-industry contexts), governance structures must align with regulatory and compliance requirements (e.g. data privacy, auditability, data residency, access controls). This implies integrating policy-as-code, auditing, data classification, consent management, and secure storage/encryption — just as any regulated enterprise system (Adabara et al., 2025; McKinsey, 2025).

10. Future Work

- Automated schema generation: Develop mechanisms where guardrail and policy definitions (e.g. allowed tool calls, output formats, safety constraints) can be automatically derived into machine-readable schemas — reducing human manual effort and minimizing human error. This approach would improve consistency and speed of guardrail deployment (Chennabasappa et al., 2025).
- Self-healing guardrails: Design guardrail layers that not only detect violations or risky behavior but can automatically adapt or repair themselves — e.g., update filters, revoke permissions, re-sandbox agents — when new patterns of misuse or drift are detected (Wang et al., 2025).
- Learning-based safety prediction: Use data-driven and ML-based methods to predict the risk profile of agent behavior before execution (or early in execution), rather than relying solely on static rules — enabling probabilistic forecasting of unsafe trajectories and pre-emptive mitigation (Zhan et al., 2025).
- Integration with sovereign-cloud and compliance frameworks: Extend guardrail and verification architectures to support regulated or highly-controlled environments (e.g. sovereign cloud deployments, compliance regimes, data residency, encryption/segregation)—ensuring that agentic AI can be safely adopted even under strict regulatory or policy constraints (IBM, 2025; Industry-governance survey, 2025).
- Formal verification frameworks for agent behavior: Adopt or build formal methods (e.g., temporal logic, model checking) to rigorously specify safety properties and verify agent plans, actions, and possible execution trajectories — particularly for embodied or high-stakes agents — so that compliance isn't just heuristic, but provably guaranteed (Zhan et al., 2025).

11. Conclusion

- We proposed a blueprint for combining deterministic guardrails (schemas, RBAC, sandboxing), runtime monitoring, and adaptive safety controls to govern agentic AI operating on an MCP foundation.

- The use of schema-driven guardrails — where agent actions, inputs/outputs, permissions and constraints are formally defined — is critical to making safety enforceable, auditable, and maintainable, especially in multi-tenant or regulated contexts.
- An MCP-based infrastructure (with strong identity, isolation, logging, compliance support) provides an ideal foundation for safe agentic AI deployment. When combined with layered guardrails and continuous verification, it enables organizations to harness automation and autonomy without sacrificing security, compliance, or control.

REFERENCES:

1. Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., & Mané, D. (2016). Concrete Problems in AI Safety. arXiv:1606.06565. (pp.2–6). https://arxiv.org/pdf/1606.06565.pdf
2. Anthropic. (2024, Nov 25). Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol
3. "AI Agent Security Best Practices." (2025). Wiz.ai Academy. https://www.wiz.io/academy/ai-agent-security?utm_source=chatgpt.com
4. Chennabasappa, S., Nikolaidis, C., Song, D., Molnar, D., Ding, S., Wan, S., & Saxe, J. (2025). LlamaFirewall: An Open Source Guardrail System for Building Secure AI Agents. arXiv preprint. https://arxiv.org/abs/2505.03574
5. Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. Communications of the ACM, 12(10), 576–580. DOI: 10.1145/363235.363259
6. IBM. (2025). What Are AI Guardrails? IBM Think. https://www.ibm.com/think/topics/ai-guardrails?utm_source=chatgpt.com
7. Koohestani, R. (2025). AgentGuard: Runtime Verification of AI Agents. arXiv preprint. https://arxiv.org/abs/2509.23864
8. Lamport, L. (2003). Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. https://lamport.azurewebsites.net/tla/book-02-08-08.pdf
9. LangChain. Tools (documentation). https://docs.langchain.com/oss/python/langchain/tools
10. JSON Schema. Draft-07 / Specification. https://json-schema.org/draft-07
11. Model Context Protocol (MCP) — Documentation and Architecture. https://modelcontextprotocol.io/docs/learn/architecture
12. MCP community discussion (schema/validator interoperability). https://github.com/modelcontextprotocol/modelcontextprotocol/discussions/1196
13. NIST. SP 800-162: Guide to Attribute-Based Access Control (ABAC). (2014). p.ix. https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-162.pdf
14. OpenAI. Function calling – OpenAI API docs / updates. https://platform.openai.com/docs/guides/function-calling
15. OpenAPI Initiative. OpenAPI Specification v3. https://spec.openapis.org/oas/v3.0.0.html
16. GraphQL Foundation. GraphQL Specification. https://spec.graphql.org/October2015/
17. gRPC. Introduction / Core concepts. https://grpc.io/docs/what-is-grpc/introduction/
18. Sandhu, R. S., Coyne, E. J., Feinstein, H. L., & Youman, C. E. (1996). Role-Based Access Control Models. IEEE Computer, 29(2), 38–47. https://csrc.nist.gov/csrc/media/projects/role-based-access-control/documents/sandhu96.pdf
19. Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. arXiv:2302.04761. https://arxiv.org/pdf/2302.04761.pdf
20. Wang, H., Poskitt, C. M., Sun, J., & Wei, J. (2025). Pro2Guard: Proactive Runtime Enforcement of LLM Agent Safety via Probabilistic Model Checking. arXiv preprint. https://arxiv.org/abs/2508.00500
21. Wang, Y., Wang, X., Yao, Y., Li, X., Teng, Y., Ma, X., & Wang, Y. (2025). SafeEvalAgent: Toward Agentic and Self-Evolving Safety Evaluation of LLMs. arXiv preprint. https://arxiv.org/abs/2509.26100

22. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2022). ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629. (p.1). https://arxiv.org/pdf/2210.03629.pdf
23. Zhan, S. S., Liu, Y., Wang, P., Wang, Z., Wang, Q., Ruan, Z., & Zhang, X. (2025). SENTINEL: A Multi-Level Formal Framework for Safety Evaluation of LLM-based Embodied Agents. arXiv preprint. https://arxiv.org/abs/2510.12985
24. Adabara, I., Sadiq, B. O., Shuaibu, A. N., Danjuma, Y. I., & Maninti, V. (2025). Trustworthy agentic AI systems: A cross-layer review of architectures, threat models, and governance strategies for real-world deployment. F1000Research. https://f1000research.com/articles/14-905?utm_source=chatgpt.com
25. “Agentic AI Security: Risks & Governance for Enterprises.” (2025). McKinsey & Company. https://www.mckinsey.com/capabilities/risk-and-resilience/our-insights/deploying-agentic-ai-with-safety-and-security-a-playbook-for-technology-leaders?utm_source=chatgpt.com
26. “Safeguarding Large Language Models: A Survey.” (2025). Artificial Intelligence Review, 58. https://link.springer.com/article/10.1007/s10462-025-11389-2?utm_source=chatgpt.com