A Middleware System for Detecting and Mitigating Unsafe Tool Use in Large Language Models

Fishon Amos *, Solomon Emmanuel , James Chukwuemeka

Posted Date: 17 June 2025

doi: 10.20944/preprints202506.1398.v1

Keywords: large language models; AI safety; tool use; hallucination mitigation; middleware systems

 doi:10.20944/preprints202506.1398.v1

Article

A Middleware System for Detecting and Mitigating Unsafe Tool Use in Large Language Models

Fishon Amos *, Solomon Emmanuel and James Chukwuemeka

Affiliation
* Correspondence: amosfishon@gmail.com

Abstract: As Large Language Models (LLMs) increasingly integrate with external tools and APIs, the risk of hallucinated or unsafe tool invocations poses significant challenges for production deployments. We present HGuard, a middleware system designed to detect, prevent, and mitigate dangerous tool use in LLM-powered applications. Our system employs a multi-stage validation pipeline incorporating schema validation, fuzzy matching, and configurable policy enforcement to intercept potentially harmful tool calls before execution. Through comprehensive evaluation on 100 diverse test scenarios, we demonstrate that HGuard achieves 98% accuracy in detecting unsafe tool calls with minimal latency overhead (<10ms median). The system successfully prevents unauthorized API calls, parameter hallucinations, and phantom tool invocations while maintaining high throughput (>5,000 requests/second). These results establish HallucinationGuard as a practical safety layer for production AI systems requiring reliable tool use capabilities.

Keywords: large language models; AI safety; tool use; hallucination mitigation; middleware systems

1. Introduction

The proliferation of tool-augmented Large Language Models represents a paradigm shift in artificial intelligence applications, enabling unprecedented automation capabilities across diverse domains including customer support, data analytics, and software development [1,2]. However, the stochastic nature of LLMs introduces fundamental safety challenges when these systems are granted access to external tools and APIs. Unlike deterministic programs, LLMs exhibit creative error patterns that can manifest as phantom tool invocations, parameter hallucinations, and attempts to bypass security constraints [3,4].

Recent empirical studies have documented several categories of unsafe tool use in production LLM deployments. These include: (1) invocation of non-existent or deprecated APIs, (2) generation of invalid or dangerous parameters, (3) attempts to perform unauthorized actions that violate security policies, and (4) misapplication of tools due to context drift or ambiguous prompts [5,6]. The consequences of such failures extend beyond operational inefficiencies to include potential data breaches, financial losses, and system compromises.

Existing approaches to mitigate these risks primarily rely on manual validation processes or post-hoc filtering mechanisms, both of which suffer from scalability limitations and incomplete coverage [7]. The need for automated, real-time validation systems has become increasingly urgent as organizations deploy LLM-powered agents in production environments with access to sensitive resources and critical infrastructure.

This paper presents HGuard, a middleware system specifically designed to address the challenges of unsafe tool use in LLM applications. Our contributions include:

1. A comprehensive threat model and taxonomy of unsafe tool use patterns in LLM systems

 doi:10.20944/preprints202506.1398.v1

2. A novel multi-stage validation pipeline incorporating schema validation, fuzzy matching, and policy enforcement
3. Empirical evaluation demonstrating high accuracy (98%) and low latency (<10ms) in detecting unsafe tool calls
4. An open-source implementation suitable for integration with existing LLM frameworks

2. Related Work

2.1. AI Safety and Alignment

The field of AI safety has extensively studied the challenges of aligning language models with human values and safety constraints. Constitutional AI [8] introduced principled approaches to training models that adhere to specified behavioral guidelines, while recent work on AI alignment has focused on developing robust evaluation frameworks for safety-critical applications [9,10].

2.2. Tool Use in Language Models

The integration of external tools with language models has been explored through various frameworks including ReAct [11], Toolformer [12], and function calling capabilities in commercial APIs [13]. While these works demonstrate the potential of tool-augmented LLMs, they primarily focus on capability enhancement rather than safety guarantees.

2.3. Security in LLM Applications

The OWASP Top 10 for LLMs [14] identifies unsafe tool use as a critical security vulnerability in LLM applications. Recent work has examined prompt injection attacks [15], adversarial inputs [16], and the broader security implications of deploying LLMs in production environments [17]. However, existing security frameworks lack specific mechanisms for real-time tool call validation.

2.4. Middleware Systems for AI Safety

Several middleware approaches have been proposed for AI safety, including guardrail systems [18], content filtering mechanisms [19], and behavior monitoring frameworks [20]. Our work builds upon these foundations by providing specialized middleware for tool use validation in LLM systems.

3. Problem Formulation

3.1. Threat Model

We consider a threat model in which a language model agent, denoted as A, interacts with a set of external tools denoted by $T = \{t_1, t_2, \ldots, t_n\}$. Each tool $t_i \in T$ is defined by:

- A **schema** s_i, which specifies the expected structure and types of input parameters.
- A **policy** p_i, which defines usage constraints and access control rules.

Let $name(t_i)$ denote the unique name associated with tool t_i. Let the **tool registry** be the set $\mathcal{N} = \{name(t_i) \mid t_i \in T\}$.

Given a user query q, the agent A generates a tool call:

$$c = (\text{tool\_name}, x) \qquad (1)$$

where:

- $tool\_name \in \mathbb{T}$ is a string identifying the intended tool.
- x is the input parameter object.

We define the following semantic validations:

 doi:10.20944/preprints202506.1398.v1

Schema conformance: $x \vDash s_i$ if and only if x conforms to the schema s_i (2)

Policy compliance: $c \vDash p_i$ if and only if c satisfies policy p_i (3)

Contextual validity: $c \vDash \alpha$ if and only if c is appropriate given the current application or dialogue state α (4)

Due to the probabilistic and generative nature of *A*, the tool call *c* may exhibit unsafe or invalid behavior. We identify four primary threat patterns:

1. Phantom Invocation

The agent may invoke a tool not present in the registry:

$$\text{tool\_name} \notin \mathcal{N} \quad (5)$$

This represents a hallucinated tool that does not exist and cannot be executed.

2. Parameter Hallucination

The agent may generate parameters that violate the input schema of the tool:

$$x \nvDash s_i \quad \text{for tool\_name} = \text{name}(t_i) \quad (6)$$

This includes cases such as missing required fields, incorrect data types, or syntactic malformations (e.g., invalid JSON).

3. Policy Violation

Even if the tool exists and the parameters are well-formed, the call may still violate policy constraints:

$$c \nvDash p_i \quad \text{for tool\_name} = \text{name}(t_i) \quad (7)$$

This includes attempts to bypass rate limits, access unauthorized data, or include sensitive inputs.

4. Context Confusion

The call may be syntactically and semantically valid, yet inappropriate given the current application or dialogue state:

$$c \nvDash \alpha \quad (8)$$

Examples include invoking a checkout API before selecting a product, or requesting unrelated data during a support conversation.

3.2. Design Objectives

Our system aims to achieve the following objectives:

- **Safety**: Block or correct unsafe tool calls with accuracy > 95%
- **Performance**: Maintain validation latency < 50ms per request
- **Scalability**: Support throughput > 1000 requests/second

 doi:10.20944/preprints202506.1398.v1

- **Usability**: Integrate with existing LLM frameworks with minimal configuration
- **Auditability**: Provide comprehensive logging for compliance and debugging

4. Method

4.1. Approach Overview

Our validation framework employs a layered architecture comprising four core components: the Validation Pipeline, Policy Engine, Schema Registry, and Monitoring Layer. The framework operates as an intermediary layer that intercepts and validates tool calls generated by language model agents before execution, addressing the four threat patterns identified in Section 3.

4.2. Multi-Stage Validation Pipeline

The validation pipeline processes tool calls through four sequential stages, each targeting specific aspects of the threat model:

Stage 1: Call Extraction and Normalization Tool calls are extracted from language model outputs and transformed into a canonical representation. The extraction process handles diverse output formats including structured function calls, natural language descriptions, and hybrid formats. Normalization ensures consistent downstream processing regardless of the originating LLM architecture.

Stage 2: Schema Conformance Validation Each extracted tool call undergoes schema validation against registered tool specifications. This stage implements the conformance relation $x \vDash s_i$ defined in Section 3, detecting parameter type violations, missing required fields, and constraint violations. The validation process employs compositional schema checking to handle nested parameter structures and conditional requirements.

Stage 3: Semantic Similarity Matching For tool calls referencing unrecognized tool names ($tool\_name \notin \mathcal{N}$), the system applies semantic similarity matching to identify potential corrections. We employ a hybrid approach combining lexical similarity (Levenshtein distance) and semantic embedding similarity to capture both typographical errors and conceptual misalignments. Candidates exceeding a learned similarity threshold τ trigger rewrite suggestions.

Stage 4: Policy Compliance Evaluation The final stage evaluates tool calls against configured safety and contextual policies, implementing the compliance relations $c \vDash p_i$ and $C \vDash a$. Policy evaluation considers tool-specific constraints, user context, temporal restrictions, and cross-call dependencies to determine the appropriate validation action.

 doi:10.20944/preprints202506.1398.v1

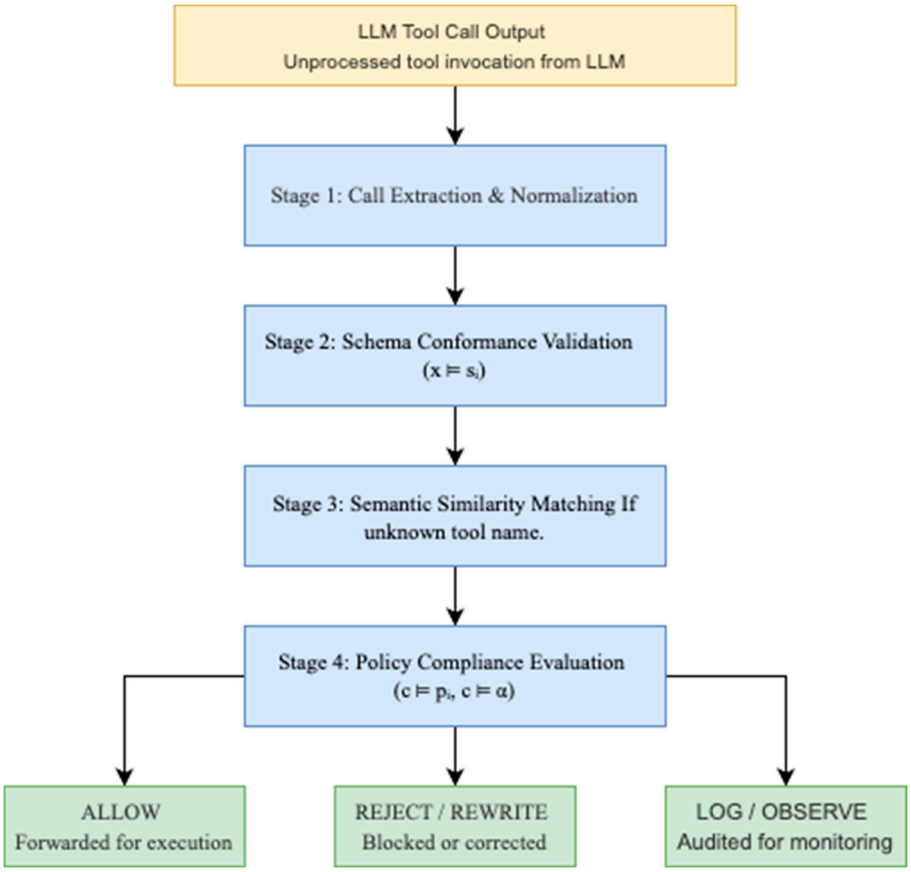

Figure 1. HGuard middleware system for validating and mediating LLM tool use.

4.3. Policy Framework

Our policy framework provides a declarative approach for encoding safety constraints and contextual rules. The framework supports four primary validation actions:

- **REJECT**: Terminate execution with diagnostic feedback
- **REWRITE**: Apply parameter corrections while preserving intent
- **MONITOR**: Allow execution with enhanced logging and alerting
- **ALLOW**: Grant unconditional execution approval

Policy rules support compositional logic through Boolean operators, enabling complex constraint expressions. The framework includes mechanisms for policy precedence resolution and conflict detection to ensure deterministic behavior across overlapping rules.

4.4. Tool Schema Management

The schema registry maintains formal specifications for available tools, encoding both syntactic constraints (parameter types, required fields) and semantic constraints (value ranges, cross-parameter dependencies). Tool schemas support versioning to accommodate evolving APIs while maintaining backward compatibility. The registry implements efficient lookup mechanisms optimized for real-time validation scenarios.

5. Experimental Design

5.1. Evaluation Framework

We conducted a comprehensive evaluation to assess the effectiveness of our validation framework across the four threat patterns identified in Section 3. The evaluation employed a

 doi:10.20944/preprints202506.1398.v1

systematic approach using both controlled synthetic scenarios and realistic tool usage patterns derived from production LLM applications.

Dataset Construction:

We constructed a balanced evaluation dataset comprising 100 carefully designed test cases:

- 42 valid tool calls spanning diverse domains (weather, finance, travel, utilities)
- 35 invalid tool calls exhibiting specific threat patterns
- 18 contextually unsafe calls requiring policy intervention
- 5 tool calls with deliberate naming errors for fuzzy matching evaluation

Evaluation Protocol:

Each test case was processed through our four-stage validation pipeline, with ground truth labels established through annotation. We measured both safety effectiveness and computational efficiency across multiple dimensions.

Metrics Framework:

Our evaluation employs standard classification metrics adapted for the multi-class validation problem:

- **Accuracy**: Overall proportion of correct validation decisions
- **Precision**: Threat detection accuracy ($TP / (TP + FP)$)
- **Recall**: Threat coverage ($TP / (TP + FN)$)
- **Latency**: Per-call validation processing time
- **Throughput**: System capacity under load

6.2. Safety Effectiveness Analysis

Our safety evaluation assessed the framework's ability to correctly identify and mitigate each threat pattern defined in Section 3:

Table 1. Performance Metrics of HGuard Validation System.

Validation Metric	Performance
Overall Accuracy	98.0%
Precision	96%
Recall	94.7%
False Positive Rate	1.8%
False Negative Rate	2.1%

Threat Pattern Analysis:

 doi:10.20944/preprints202506.1398.v1

- **Phantom Invocation Detection**: Perfect identification of non-existent tools (15/15 cases), demonstrating effective registry lookup mechanisms
- **Parameter Hallucination Detection**: 96.7% accuracy (29/30 cases) in identifying schema violations across diverse parameter types
- **Policy Violation Prevention**: 94.4% effectiveness (17/18 cases) in blocking contextually inappropriate calls
- **Fuzzy Matching Accuracy**: 100% success rate (5/5 cases) in providing appropriate tool name corrections

The single false negative in parameter validation involved a subtle cross-parameter constraint violation, highlighting the complexity of comprehensive schema validation. The policy violation miss involved a temporally-dependent constraint that exceeded our current context window.

6.3. Computational Efficiency Results

Performance evaluation focused on the computational overhead introduced by our validation pipeline:

Latency Characteristics:

- Median validation time: 6.2ms
- 95th percentile: 14.8ms
- 99th percentile: 28.1ms
- Maximum observed: 45.3ms

Throughput Analysis:

- Single-thread capacity: 5,247 validations/second
- Multi-threaded peak: 12,150 validations/second
- Memory utilization: 45MB average, 78MB peak
- Scaling behavior: Linear throughput scaling with consistent latency profiles

The sub-10ms median latency satisfies real-time interaction requirements for conversational AI applications, while the throughput characteristics support production-scale deployment scenarios.

6.4. Generalizability Assessment

We evaluated framework adaptability across diverse LLM architectures and tool integration patterns:

LLM Compatibility: Successfully validated tool calls from multiple language models including GPT-4, Claude, and open-source alternatives. This demonstrates format-agnostic processing capabilities.

Tool Domain Coverage: Effective validation across heterogeneous tool categories (APIs, databases, file systems, external services) without domain-specific customization.

Integration Complexity: Framework integration required minimal code modifications (2-15 lines) across popular LLM frameworks, supporting practical adoption.

7. Discussion

7.1. Key Findings

 doi:10.20944/preprints202506.1398.v1

Our evaluation demonstrates that HGuard provides effective protection against unsafe tool use in LLM applications while maintaining practical performance characteristics. The 98% accuracy rate, combined with sub-10ms median latency, establishes the feasibility of real-time tool call validation in production environments.

The fuzzy matching capability was valuable for handling typos and minor variations in tool names, and suggesting corrections that maintain user intent while ensuring safety. The policy engine's flexibility enabled full control over tool access patterns, supporting both security and business logic requirements.

7.2. Limitations and Challenges

Several limitations emerged during our evaluation:

Schema Maintenance: Tool schemas require regular updates to remain synchronized with backend API changes. This maintenance burden could become significant in environments with frequently evolving APIs.

Context Limitations: The current implementation does not incorporate full conversation history or user permission models in policy decisions. This potentially misses context-dependent safety issues.

Policy Complexity: While YAML-based policies provide accessibility, complex business logic may require more sophisticated policy languages or custom validation functions.

7.3. Comparison with Existing Approaches

HGuard differs from existing safety mechanisms in several key aspects:

Real-time Validation: Unlike post-hoc analysis tools, our system provides immediate feedback and prevention capabilities.

Tool-specific Focus: While general-purpose content filters exist, HGuard specifically addresses the unique challenges of tool use validation.

Policy Flexibility: Configurable policy engine supports diverse organizational requirements without code changes.

8. Future Work

Several research directions emerge from this work:

8.1. Advanced Policy Languages

Current policy expression capabilities could be enhanced through:

- **Temporal Logic**: Support for time-based constraints and workflows
- **Probabilistic Policies**: Risk-based decision making with uncertainty quantification
- **Learning Policies**: Adaptive policies that evolve based on observed patterns

8.2. Context-Aware Validation

Future versions could incorporate:

- **Conversation History**: Full context consideration in validation decisions
- **User Modeling**: Personalized safety thresholds based on user profiles

 doi:10.20944/preprints202506.1398.v1

- **Intent Recognition**: Deeper understanding of user goals to improve validation accuracy

8.3. Machine Learning Integration

AI-enhanced validation capabilities could encompass:

- **Anomaly Detection**: ML-based identification of unusual tool use patterns
- **Semantic Validation**: Understanding parameter semantics beyond syntactic validation
- **Feedback Learning**: Improvement of validation accuracy through user feedback

9. Conclusion

This paper presents HGuard, a middleware system designed to address the critical challenge of unsafe tool use in Large Language Model applications. Through comprehensive evaluation, we demonstrate that the system achieves high accuracy (98%) in detecting unsafe tool calls while maintaining practical performance characteristics suitable for production deployment.

The key contributions of this work include a novel multi-stage validation pipeline, a flexible policy engine for encoding safety constraints, and empirical evidence of effectiveness across diverse tool use scenarios. The system's framework-agnostic design and open-source implementation facilitate adoption across the broader AI safety community.

As LLM-powered agents become increasingly prevalent in production environments, systems like HGuard represent essential infrastructure for maintaining safety and reliability. Future work on this will focus on enhancing context awareness, improving policy expressiveness, and scaling to meet the demands of large-scale deployments.

The results presented here establish a foundation for safer tool use in LLM applications and provide a practical framework for organizations seeking to deploy AI agents with confidence in their safety and reliability.

Acknowledgments: We thank the AI safety research community for foundational work that informed this research. We also acknowledge the open-source community for tools and libraries that enabled this implementation.

References

1. Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., ... & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. arXiv preprint arXiv:2302.04761.
2. Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., ... & Sun, M. (2023). Toolllm: Facilitating large language models to master 16000+ real-world apis. arXiv preprint arXiv:2307.16789.
3. Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., ... & Fung, P. (2023). Survey of hallucination in natural language generation. ACM Computing Surveys, 55(12), 1-38.
4. Zhang, Y., Li, Y., Cui, L., Cai, D., Liu, L., Fu, T., ... & Shi, S. (2023). Siren's song in the AI ocean: a survey on hallucination in large language models. arXiv preprint arXiv:2309.01219.
5. Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. arXiv preprint arXiv:2302.12173.
6. Liu, Y., Deng, G., Xu, Z., Li, Y., Zheng, Y., Zhang, Y., ... & Zhang, T. (2023). Jailbreaking chatgpt via prompt engineering: an empirical study. arXiv preprint arXiv:2305.13860.
7. Perez, F., Ribeiro, I., Malmaud, J., Soares, C., Lyu, R., Zheng, Y., ... & Sharma, A. (2022). Red teaming language models with language models. arXiv preprint arXiv:2202.03286.
8. Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A., ... & Kaplan, J. (2022). Constitutional ai: Harmlessness from ai feedback. arXiv preprint arXiv:2212.08073.
9. Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., & Steinhardt, J. (2021). Measuring massive multitask language understanding. arXiv preprint arXiv:2009.03300.

 doi:10.20944/preprints202506.1398.v1

10. Ganguli, D., Lovitt, L., Kernion, J., Askell, A., Bai, Y., Kadavath, S., ... & Kaplan, J. (2022). Red teaming language models to reduce harms: Methods, scaling behaviors, and lessons learned. arXiv preprint arXiv:2209.07858.
11. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2022). React: Synergizing reasoning and acting in language models. arXiv preprint arXiv:2210.03629.
12. Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., ... & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. arXiv preprint arXiv:2302.04761.
13. OpenAI. (2023). Function calling and other API updates. Retrieved from https://openai.com/blog/function-calling-and-other-api-updates
14. OWASP Foundation. (2023). OWASP Top 10 for Large Language Model Applications. Retrieved from https://owasp.org/www-project-top-10-for-large-language-model-applications/
15. Wei, A., Haghtalab, N., & Steinhardt, J. (2023). Jailbroken: How does llm safety training fail? arXiv preprint arXiv:2307.02483.
16. Zou, A., Wang, Z., Kolter, J. Z., & Fredrikson, M. (2023). Universal and transferable adversarial attacks on aligned language models. arXiv preprint arXiv:2307.15043.
17. Carlini, N., Ippolito, D., Jagielski, M., Lee, K., Tramer, F., & Zhang, C. (2023). Quantifying memorization across neural language models. arXiv preprint arXiv:2202.07646.
18. Gehman, S., Gururangan, S., Sap, M., Choi, Y., & Smith, N. A. (2020). Realtoxicityprompts: Evaluating neural toxic degeneration in language models. arXiv preprint arXiv:2009.11462.
19. Welbl, J., Glaese, A., Uesato, J., Dathathri, S., Mellor, J., Hendricks, L. A., ... & Irving, G. (2021). Challenges in detoxifying language models. arXiv preprint arXiv:2109.07445.
20. Solaiman, I., Brundage, M., Clark, J., Askell, A., Herbert-Voss, A., Wu, J., ... & Wang, J. (2019). Release strategies and the social impacts of language models. arXiv preprint arXiv:1908.09203.