VeriGuard: Enhancing LLM Agent Safety via Verified Code Generation

Anonymous authors
Paper under double-blind review

Abstract

The deployment of autonomous AI agents in sensitive domains, such as healthcare, introduces critical risks to safety, security, and privacy. These agents may deviate from user objectives, violate data handling policies, or be compromised by adversarial attacks. Mitigating these dangers necessitates a mechanism to formally guarantee that an agent's actions adhere to predefined safety constraints, a challenge that existing systems do not fully address. We introduce VeriGuard, a novel framework that provides formal safety guarantees for LLM-based agents through a dual-stage architecture designed for robust and verifiable correctness. The initial offline stage involves a comprehensive validation process. It begins by clarifying user intent to establish precise safety specifications. VeriGuard then synthesizes a behavioral policy and subjects it to both extensive testing in simulated environments and rigorous formal verification to mathematically prove its compliance with these specifications. This iterative process refines the policy until it is deemed correct. Subsequently, the second stage provides online action monitoring, where VeriGuard operates as a runtime monitor to validate each proposed agent action against the pre-verified policy before execution. This separation of the exhaustive offline validation from the lightweight online monitoring allows formal guarantees to be practically applied, providing a robust safeguard that substantially improves the trustworthiness of LLM agents in complex, real-world environments.

1 Introduction

The proliferation of Large Language Model (LLM) agents marks a significant leap towards autonomous AI systems capable of executing complex, multi-step tasks (Xi et al., 2023; Yao et al., 2023). These agents, often empowered to interact with external tools, APIs, and file systems (Schick et al., 2023a; Patil et al., 2024), hold immense promise for automating digital workflows and solving real-world problems. However, this power introduces substantial and often unpredictable safety and security vulnerabilities. A critical reliability gap has emerged: while LLM agents can generate solutions with unprecedented flexibility, the solution they produce often lacks assurances, making it susceptible to subtle errors, security flaws, and emergent behaviors that can lead to catastrophic failures. An agent tasked with data analysis could inadvertently exfiltrate sensitive information; one managing cloud infrastructure could execute destructive commands; another interacting with financial APIs could trigger erroneous, irreversible transactions. This problem is even more serious when there is adversary attack on the system, as shown in Zhang et al. (2025).

Existing safety mechanisms—such as sandboxing, input/output filtering, and static rule-based guardrails (Inan et al., 2023; Rebedea et al., 2023) —provide a necessary but insufficient first line of defense. These approaches are fundamentally reactive or based on pattern matching; they struggle to cover the vast and dynamic state space of agent actions and can be bypassed by novel adversarial inputs or unforeseen edge cases (Wei et al., 2023; Xu et al., 2023). They lack a deep, semantic understanding of the code's intent and consequences, treating the agent's output as a black box to be constrained. This leaves systems vulnerable to sophisticated exploits that a static rule set cannot anticipate (Schulhoff et al., 2023). For LLM agents to be trusted in high-stakes, mission-critical environments, a more rigorous, provable approach to safety is required.

In this work, we propose a novel method to address this reliability gap, centered on the VeriGuard framework. VeriGuard represents a paradigm shift from reactive filtering to proactive, provable safety by deeply integrating policy specification generation and automated verification into the agent's action-generation pipeline. VeriGuard fundamentally reshapes the code generation process to be "correct-by-construction". This is achieved by prompting the LLM agent to generate not only the functional code for an action but also its corresponding verification that precisely define the code's expected behavior and safety properties. These paired artifacts are then immediately subjected to an automated verification engine. An iterative refinement loop forms the core of our framework: if verification fails, the verifier provides a specific counterexample or logical inconsistency, which is fed back to the agent as a concrete, actionable critique to guide the generation of a corrected and verifiably safe version of the code (Pan et al., 2024; Zhao et al., 2025). More details are in §3.

The primary contribution of this paper is the VeriGuard framework itself, which includes novel methodologies for the LLM-driven generation and refinement of verifiable code tailored to agent security and safety contexts. We further contribute a comprehensive empirical validation of the framework's effectiveness in preventing unsafe actions across a variety of challenging domains. Finally, we present a detailed analysis of the performance trade-offs inherent in this approach.

2 Related Work

2.1 LLM Agents and the Emergence of Autonomous Systems

The development of Large Language Models (LLMs) has catalyzed the emergence of a new class of autonomous systems known as LLM agents. LLM agents are designed to be proactive, goal-oriented entities capable of planning, reasoning, and interacting with their environment through the use of tools (Schick et al., 2023b). Early frameworks like ReAct demonstrated how to synergize reasoning and acting within LLMs, enabling them to solve complex tasks by generating both textual reasoning traces and executable actions (Yao et al., 2023). The agent can also execute more complex tasks like web browsing. This capability, however, is merely the entry point into a broader spectrum of autonomous actions. Advanced agents are not just navigating websites but are becoming generalist problem-solvers on the web and beyond. This evolution is detailed in research and demonstrated in benchmarks like WebArena (Zhou et al., 2023) and Mind2Web (Gou et al., 2025), which test agents on their ability to perform multi-step, realistic tasks on live websites.

This paradigm quickly evolved into more sophisticated agent architectures. Systems like AutoGPT (Gravitas, 2023) and BabyAGI showcased the potential for fully autonomous task completion, where agents could decompose high-level goals into smaller, executable steps, manage memory, and self-direct their workflow. Further research has explored enhancing agent capabilities through mechanisms like self-reflection and verbal reinforcement learning, allowing them to learn from past mistakes and improve their performance over time (Shinn et al., 2023). The concept of "Generative Agents" pushed the boundaries even further by creating interactive simulacra of human behavior within a sandbox environment, highlighting the potential for complex social and emergent behaviors (Park et al., 2023). A comprehensive survey by (Wang et al., 2023) details the rapid advancements and architectural patterns in this burgeoning field.

2.2 LLM Safety, Alignment, and Guardrails

A significant body of research has focused on ensuring the safety and alignment of LLMs. A primary line of defense involves creating guardrails to constrain agent behavior. These can range from simple input/output filtering and prompt-based restrictions to more sophisticated techniques (Bai et al., 2022). Another critical area is the proactive discovery of vulnerabilities through "red teaming", where humans or other AIs craft adversarial prompts to elicit unsafe or undesirable behaviors from the model (Ganguli et al., 2022). The insights from these attacks are then used to fine-tune the model for greater robustness. Despite these efforts, LLMs remain susceptible to a wide array of "jailbreaking" techniques that can bypass safety filters (Wei et al., 2023). More recent work has focused on creating safety-tuned LLMs specifically for tool use, aiming to prevent harmful API calls or command executions (Jin et al., 2024).

There are some previous work in Agent safeguard. GuardAgent, a framework that uses an LLM-based "guard agent" to safeguard other LLM agents. GuardAgent operates as a protective layer, using reasoning to detect and prevent unsafe behaviors (Xiang et al., 2025). Another work is ShieldAgent, a guardrail agent designed to ensure that autonomous agents powered by large language models (LLMs) adhere to safety policies (Chen et al., 2025).

However, these existing approaches are largely empirical and reactive. They rely on identifying and patching vulnerabilities as they are discovered, but they do not provide formal, provable guarantees of safety. A clever adversary can often devise a novel attack that circumvents existing guardrails. This highlights a fundamental limitation: without a formal specification of what constitutes "safe" behavior and a method to verify compliance, safety remains an ongoing. VeriGuard distinguishes itself from this body of work by moving from an empirical to a formal verification paradigm, aiming to prove the correctness of an agent's actions before they are ever executed.

2.3 Formal Methods and Verifiable Code Generation

Formal methods provide a mathematically rigorous set of techniques for the specification, development, and verification of software and hardware systems. The advent of powerful LLMs has opened a new frontier for bridging the gap between natural language specifications and formal, machine-checkable code. Recent research has begun to explore the potential for LLMs to automate or assist in the generation of not just code, but also its formal specification and verification artifacts. For example, (Li et al., 2024) demonstrate a system where LLMs are used to generate verifiable computation, producing code along with the necessary components for a verifier to check its correctness. Further studies have investigated the self-verification capabilities of LLMs (Ghaffarian et al., 2024). This line of work shows the promise of integrating LLMs into high-assurance software development pipelines.

3 Methodology

Figure 1 describes the high-level ideas of VeriGuard, which operates in two main stages: **(i) Policy Generation**: VeriGuard takes inputs including the agent's specification and a high-level security request in natural language to synthesize an initial policy function and its corresponding formal constraints. To ensure the correctness and alignment of this policy, we employ a rigorous, multi-step refinement feedback loop. This loop begins with a validation phase to resolve any ambiguities in the user's request, followed by an automated code testing phase that generates unit tests to verify functional correctness. The most critical phase uses formal verification to prove that the policy code adheres to its specified conditions, ensuring a provably-sound safety contract.. **(ii) Policy Enforcement**: The verified policy is integrated into the agentic system at key enforcement points, where it intercepts and evaluates agent-initiated actions before execution. When a potential violation is detected, VeriGuard can employ one of several enforcement strategies, ranging from immediately terminating the agent's task to blocking the specific unsafe action or engaging in a collaborative re-planning dialogue with the agent.

3.1 Task Definition

In this section, we formalize the process of generating agent policies from high-level, natural language specifications.

Policy Generation Given a safety and security request in natural language, denoted as r, and a agent specification, $\mathcal{S}$, the primary objective is to synthesize a policy function, p, written in a structured programming language. Concurrently, a set of verifiable constraints (i.e. pre and post-conditions), $C = \{c_1, c_2, \ldots, c_n\}$, is derived. The system must guarantee that the generated policy p complies with all constraints in C. This relationship is formally denoted as $p \models C$, signifying that $\forall c \in C$, the policy p satisfies c. The user request r typically defines a security or operational protocol in text format, while the agent specification $\mathcal{S}$

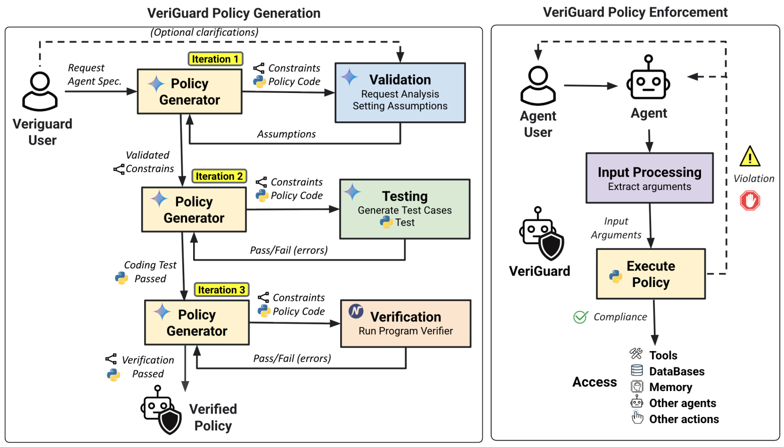

Figure 1: VERIGUARD overview which includes Policy generation and Policy enforcement. The verified policy is integrated into the agent as a runtime safeguard, intercepting and preventing harmful actions.

provides a schematic of the agent: task description, input/output (I/O) data structures, available context, environmental information, and any other available data.

Policy Enforcement Given an agentic system and a set of verified policies, the second objective is to integrate these policies as enforcement mechanisms. The goal is to optimize the system's performance by minimizing policy violations (i.e., reducing the attack surface) while maximizing the agent's task-completion utility.

3.2 FRAMEWORK

To address the defined tasks, we propose a framework, **VeriGuard**, which consists of an initial policy generator followed by an iterative refinement loop. This loop validates, tests, and formally verifies the policy code to ensure it accurately reflects the agent requirements and specifications. For the policy enforcement task, experiment with multiple integration strategies for deploying VERIGUARD within an agentic system.

3.2.1 POLICY GENERATOR

The Policy Generator is the core component responsible for translating the agent's specification and user's intent into executable code and formal specifications. It has two sub-components: (1) policy code generation, and (2) constrains generation, both LLM-based. At the first pass, the policy generator takes the user request r, an agent specification $\mathcal{S}$, to produce a preliminary policy function p_0, together with a list of arguments the policy function requires $\mathcal{P}_0$. Similarly, the constrains generator take same inputs to produce a set of constraints C_0. This initial generation functions, G_0 and H_0, can be represented as:

$$G_0(r, \mathcal{S}) \to (p_0, \mathcal{P}_0) \qquad H_0(r, \mathcal{S}) \to (C_0)$$

The arguments schema $\mathcal{P}_0$ contains the name, description and type of each required input argument of the policy function. If the request entails multiple interdependent rules, the generator produces a single, cohesive codebase that encapsulates all logic. The prompts for the initial generations are detailed in A.1.

The Policy Generator operates within an iterative refinement loop where policy and constraints are gradually improved from the previous step (p_{t-1}, C_{t-1}):

$$G_t(r, \mathcal{S}, R, A, e, p_{t-1}) \to (p_t, \mathcal{P}_t) \qquad H_t(r, \mathcal{S}, R, A, p_{t-1}) \to (C_t)$$

R, A, and e are the set of requirements, assumptions and coding error messages.

3.2.2 REFINEMENT PROCESS

We employ a three-stage refinement process: validation, testing, and formal verification.

Validation The Validator's primary function is to resolve ambiguities and ensure the semantic alignment between the user's natural language request and its formal representation (p_0, C_0). This process is bifurcated into an analysis phase and a disambiguation phase.

In the analysis phase, a function V_a scrutinizes the initial artifacts to identify semantic ambiguities, logical inconsistencies, and implicit presuppositions. The output is a set of queries, Q, that encapsulate these issues: $V_a(p_0, C_0) \rightarrow Q$

In the disambiguation phase, a function V_d processes the user's feedback, U_{feedback}, to resolve the queries in Q. This interactive process yields a definitive set of explicit assumptions, A, and a refined, unambiguous set of requirements R as: $V_d(Q, U_{\text{feedback}}) \rightarrow (A, R)$

In an autonomous operational mode where user feedback is unavailable, an internal module, Ω, is invoked to resolve the queries by selecting the most contextually plausible interpretations. This generates a set of default assumptions, A_{default}, which are then used to produce the final requirements R. This autonomous path is modeled as: $V_d(Q, \Omega(Q)) \rightarrow (A_{\text{default}}, R)$. A.3 shows the implementations detail of this component.

Code Testing This module automatically generates a suite of test cases to perform empirical validation of the policy function. It takes the policy code p, the user request r, and the agent specification $\mathcal{S}$ as input. The objective is to ensure that the policy meets a baseline of functional requirements and correctly handles typical and edge-case scenarios before proceeding to the more computationally expensive formal verification stage. The output is a set of test cases formatted for the *PyTest* framework. The policy code is refined iteratively until all generated tests pass, with failure reports and error messages e serving as feedback for the refinement loop. The iteration stops when not more errors are found or at a maximum N number. Details in A.4.

Verification The final stage of refinement involves formal verification using a program verifier. This component takes the logical constraints C and the policy code p as input. The constraints in C define a formal contract, specifying the pre-conditions ($C_{\text{pre}} \subseteq C$) that must hold before the policy's execution and the post-conditions ($C_{\text{post}} \subseteq C$) that must be guaranteed upon its completion.

The verifier's task is to mathematically prove that the policy code p adheres to this contract. This relationship is formally expressed using a Hoare triple: $\{C_{\text{pre}}\}\ p\ \{C_{\text{post}}\}$. If program p starts in a state where pre-condition C_{pre} is true, its execution is guaranteed to terminate in a state where post-condition C_{post} is true. If the code violates the contract, the verifier provides a counterexample or error trace e, which is used as feedback to refine the policy or constraints. The refinement cycle continues until formal verification succeeds or at a maximum N number. For this implementation, we utilize the Nagini verifier (Eilers & Müller, 2018) as a black box. As a static verifier built on the Viper (Eilers et al., 2025) infrastructure, Nagini can handle more complex properties than other available Python verifiers. Pre-processing for Nagini is detailed in A.5.

3.3 Policy Enforcement Strategies

Once a policy is generated and verified, it is integrated into the agentic system at specific enforcement points that intercept agent-initiated actions (e.g., tool executions, database access, environmental interactions). Each agent can be governed by one or more policy functions.

3.3.1 Policy Function Arguments

At runtime, the arguments for the policy function defined, in $\mathcal{P}$, must be populated from the agentic system data defined in S. We do not assume S is a direct input to the policy, as this data could be unstructured, and require preprocessing or extraction. Moreover, implementing preprocessing step strictly via code can limit the system's flexibility. Thus, a function $f : S \rightarrow P$ is required to map the agent data to the policy arguments. For our experiments, we implement f as a flexible LLM-based component (A.2). The input of f is the agent data in the format specified in S and the output is a populated dictionary of arguments specified in P.

3.3.2 Policy Function Integration

We experimented with four distinct enforcement strategies upon detecting a policy violation: **(i) Task Termination:** the most restrictive approach, which halts the agent's entire high-level task and issues a notification explaining the violation; **(ii) Action Blocking:** a more targeted approach, where the specific action that violates the policy is blocked, but the agent is permitted to continue executing subsequent actions in its plan that do not violate policy; **(iii) Tool Execution Halt:** which stops the specific execution that caused the violation and returns no observation to the agent, forcing the agent's reasoning process to halt and decide on a new course of action based on the lack of feedback; and **(iv) Collaborative Re-planning:** the most interactive strategy, which informs the agent of the specific policy it has violated and requests that it generate a new action plan that complies with the stated constraints.

4 Experiments

4.1 Datasets

Agent Security Bench (ASB) (Zhang et al., 2025) This dataset focuses on four types of attacks against agentic systems: direct and indirect prompt injection, memory poisoning, and plan-of-thought backdoors. The framework utilizes 10 ReAct agents across 10 different scenarios (e.g., finance, commerce, autonomous driving), incorporating 10 different tasks, 13 attack variations, and 400 attack tools.

EICU-AC (Xiang et al., 2025) This benchmark evaluates an ICU question/answer agent's access control over 10 patient databases. It defines specific user roles, each restricted to designated subsets of the databases and their internal information categories. The agent must deny any user question if its answer requires accessing data that is forbidden for the user's given role.

Mind2Web-SC (Xiang et al., 2025) Mind2Web-SC adapts the Mind2Web benchmark, which tasks web agents with solving over 2,000 complex online tasks using sequential actions. The "SC" (Safety Control) variant augments this framework by introducing safety rules that prohibit specific users from performing certain web activities, thereby testing the agent's ability to enforce these policies.

4.2 Evaluation Metrics

ASB: we simultaneously evaluate defense efficacy and operational interference using two primary metrics: the **(i) Attack Success Rate (ASR↓)**: quantifies the quality of the defense. It is defined as the ratio of successful attacks to the total number of attack attempts. **(ii) Task Success Rate (TSR↑)**: measures the operational interference or overhead imposed by the defense mechanism. It is calculated as the ratio of successfully completed original tasks to the total number of tasks attempted.

EICU-AC and **Mind2Web-SC**: the task is defined exclusively as access control, where correctly stopping a forbidden action is the desired outcome. This evaluation is treated as a binary classification problem (Allow vs. Deny), so we use standard classification metrics: **Accuracy**, **Precision**, and **Recall**.

4.3 Baselines

ASB: we experiment with four backbone LLMs: Gemini 2.5 Flash and Pro, GPT-4.1, and Claude-Sonnet-4. We establish two base boundaries: the upper-bound *No Attack*, derived from unattacked model performance, and *No Defence*, derived from undefended model performance.We compare against several baselines: **Paraphrasing** (Jain et al., 2023), which rewords the query to disrupt malicious special-character sequences and triggers (effective for DPI and PoT Backdoor attacks); **Dynamic Prompt Rewriting** (Zhang et al., 2025), which transforms the input to align with security objectives (proposed for DPI); and **Delimiter** (Mattern et al., 2023), which encapsulates the user query to ensure bounded execution (effective for IPI). We also implemented a stronger **Guardrail** baseline that receives the same input as VeriGuard but, instead of generating a code function, directly asks an LLM to evaluate policy compliance.

EICU-AC and **Mind2Web-SC**: we report the results of several state-of-the-art (SOTA) approaches. These include **GuardAgent** (Xiang et al., 2025), which translates natural language safety rules into executable code via manually defined functions; **AGrail** (Luo et al., 2025), which implements a mechanism to continually learn and adapt policies (as security checks) and uses an LLM for verification; **LLaMA-Guard 3** Llama Team (2024), a model trained to detect security issues; and **AgentMonitor** (Chan et al., 2024), a guardrail method for multi-agent systems. We also include the **Hard-coded Rules** baseline in (Xiang et al., 2025).

4.4 Results

Table 1 summarizes our evaluation on the ASB dataset, conducted across three backbone LLMs to assess generalization. The table reports the ASR and TSR against several baselines, including a "No Defense" scenario (providing a lower bound for ASR) and a "No Attack" scenario (an upper bound for TSR).

We compare VeriGuard against *Paraphrase* and *Rewrite*, the best performing defenses from the ASB benchmark. We also introduce a stronger baseline, *GuardRail*, which, given the same inputs as VeriGuard, directly queries an LLM to determine if an action violates the user's request, rather than generating a verifiable policy.

The low ASR achieved by GuardRail indicates that simple violation detection is a largely solved task for strong LLMs. The primary challenge, therefore, is not if a violation occurs, but how to intervene precisely by blocking only the malicious component (e.g., a specific tool) without degrading task utility. VeriGuard proves particularly effective at this, achieving a near-zero ASR while simultaneously outperforming all other defenses in TSR, demonstrating a superior trade-off between security and utility.

Table 2 summarizes the performance evaluation on the EICU-AC and Mind2Web-SC datasets. To ensure a fair comparison, we use GPT-4o as the backbone LLM, consistent with the SOTA model. We also report with Gemini-2.5-pro. VeriGuard, achieves perfect accuracy on the EICU-AC dataset and outperforms all baselines on recall in Mind2Web-SC. This is particularly noteworthy given that VeriGuard is a generic policy constructor, whereas a strong baseline like GuardAgent employs a predefined policy structure specifically tailored to these access control tasks. Furthermore, unlike GuardAgent, our method does not require any in-context learning to build its policies. On the other hand, Agrail shows better accuracy and precision showing that an external memory bank of policies can be beneficial. Future, work can enhance VeriGuard with memory of previous judgments.

While our method attains high accuracy, we argue that recall is a more critical metric for security applications. On both datasets, VeriGuard achieves high recall, signifying that it successfully identifies and blocks every policy violation. This capacity to prevent all illicit actions, even at the cost of a decrease in precision, is a crucial requirement for deploying secure agentic systems.

5 Analysis

5.1 Ablation Study of VeriGuard Components

The results of our ablation study, presented in Figure 2, detail the cumulative impact of each VeriGuard component. The analysis was conducted on the Agent Security Benchmark (ASB), utilizing Gemini-2.5-Flash with default parameters.

Figure 2a shows the defense is built in stages as initially the agent is highly vulnerable, with an average ASR of 53.5%. First, the Policy Generation step provides a substantial impact, reducing the average ASR to 9.97%. Subsequently, the Validation plays a critical role for complex attacks where the initial policy may be incomplete or non-executable; this is most evident against Memory Poisoning, where this step reduces the ASR by more than half (from 31.75% to 15%). Following this, the Validation component further enhances robustness, fully neutralizing all remaining threats and reducing the ASR to 0% across all attack vectors. Finally, the formal verification step ensures that the defense code rigorously follows all security constraints.

Table 1: Experiment results of VeriGuard on ASB benchmark. Attack Success Rate (ASR ↓) Task Success Rate (TSR ↑).

Defense	**DPI**		**IPI**		**MP**		**PoT**		**AVG**	
	ASR↓	*TSR↑*	*ASR↓*	*TSR↑*	*ASR↓*	*TSR↑*	*ASR↓*	*TSR↑*	*ASR↓*	*TSR↑*
Gemini-2.5-Flash										
No attack	–	57.5	–	57.5	–	57.5	–	74.3	–	61.7
No defense	98.5	0.5	40.5	46.3	15	57.3	53.5	64.3	51.9	42.1
Delimiter	–	–	40.8	48.5	–	–	–	–	–	–
Paraphrase	71.8	24.0	–	–	–	–	57.3	67.3	–	–
Rewrite	70.5	30.0	–	–	–	–	–	–	–	–
GuardRail	0.0	24.5	0.0	35.3	0.0	58.5	0.0	66.3	0.0	40.2
VeriGuard	**0.0**	**50.5**	**0.0**	**55.8**	**0.0**	**78.5**	**0.0**	**77.7**	**0.0**	**63.3**
Gemini-2.5-Pro										
No attack	–	76.0	–	76.0	–	76.0	–	78.0	–	76.5
No defense	83.0	3.5	62.3	68.0	11.0	79.8	52.2	75.5	52.1	56.7
GuardRail	**0.0**	48.8	**0.0**	18.0	**0.0**	67.3	**0.0**	**72.0**	**0.0**	51.5
VeriGuard	**0.0**	**55.6**	**0.0**	**65.5**	**0.0**	**76.8**	**0.0**	71.3	**0.0**	**67.3**
GPT-4.1										
No attack	–	64.5	–	64.5	–	64.5	–	87.0	–	70.1
No defense	92.5	1.0	60.0	45.3	2.8	62.3	99.5	87.0	63.7	43.1
Delimiter	–	–	64.3	**52.0**	–	–	–	–	–	–
Paraphrase	80.3	19.0	–	–	–	–	60.0	85.5	–	–
Rewrite	74.5	15.5	–	–	–	–	–	–	–	–
GuardRail	**0.0**	20.0	**0.0**	31.5	**0.0**	63.0	**0.0**	82.0	**0.0**	44.6
VeriGuard	**0.0**	**28.0**	**0.0**	42.3	**0.0**	**63.5**	**0.0**	**94.5**	**0.0**	**57.1**
Claude-sonnet-4										
No attack	–	100.0	–	100.0	–	100.0	–	99.0	–	99.8
No defense	31.3	89.0	63.8	97.0	24.0	82.0	80.5	87.8	49.9	89.0
Delimiter	–	–	60.8	**98.3**	–	–	–	–	–	–
Paraphrase	39.8	**88.5**	–	–	–	–	73.3	**90.5**	–	–
Rewrite	66.8	57.5	–	–	–	–	–	–	–	–
GuardRail	**0.0**	68.5	**0.0**	46.0	**0.0**	75.5	**0.0**	83.5	**0.0**	68.3
VeriGuard	**0.0**	86.8	**0.0**	91.5	**0.0**	**81.8**	**0.0**	80.2	**0.0**	**85.1**

Table 2: Performance comparison on the EICU-AC and Mind2Web-SC benchmarks

Methods	**EICU-AC**			**Mind2Web-SC**		
	Acc.	P	R	Acc.	P	R
Hardcoded Safety Rules (Xiang et al., 2025)[a]	81.0	76.6	90.7	77.5	95.1	58.0
LLaMA-Guard3 (Luo et al., 2025)*	48.7	–	–	56.0	93.0	13.0
AgentMonitor (GPT-4o) (Luo et al., 2025)*	82.3	98.2	66.7	72.5	79.2	61.0
GuardAgent (GPT-4) (Xiang et al., 2025)*	98.7	**100.0**	97.5	90.0	**100.0**	80.0
AGrail (GPT-4o) (Luo et al., 2025)*	97.8	97.5	98.1	**98.4**	99.0	98.0
VeriGuard (GPT-4o)	**100.0**	**100.0**	**100.0**	95.1	91.3	99.0
VeriGuard (GPT-4.1)	**100.0**	**100.0**	**100.0**	96.2	91.2	**100.0**
VeriGuard (Gemini-2.5-pro)	**100.0**	**100.0**	**100.0**	97.2	95.0	99.0

[a] Values obtained from the cited papers.

Figure 2b demonstrates that these robust security gains do not incur a performance trade-off. The TSR remains high and exhibits a consistent increase (from 51.87% to 63.25% average), confirming VeriGuard's ability to secure the agent without compromising functional utility.

5.2 Evaluating Integration Methods: Security vs. Utility

Table 3 presents the results from the ASB using Gemini-2.5-Flash. It evaluates the five integration strategies detailed in Section 3.3.2: Task Termination (TT), Action Blocking

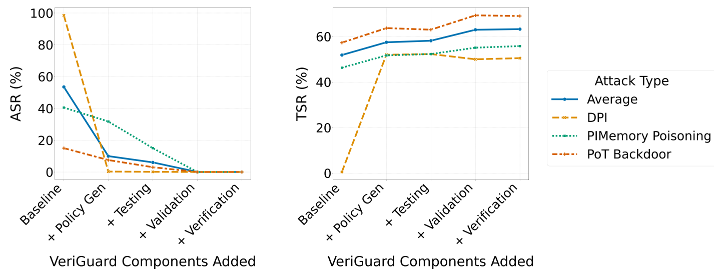

Figure 2: **(Left)** shows the ASR is systematically reduced to 0% across all evaluated attack types. **(Right)** shows the TSR increases as defense layers are added.

(AB), Tool Execution Halt (TEH), Collaborative Re-planning (CRP), and a combination of CRP and TEH.

Table 3: ASR↓ and TSR↑ results for VeriGuard integration methods including Task Termination (TT), Action Blocking (AB), Tool Execution Halt (TEH), Collaborative Re-planning (CRP). `CRP + TEH` combination achieves the optimal balance of security and utility.

Integration Method	**DPI**		**IPI**		**MP**		**PoT**		**AVG**	
	ASR↓	TSR↑	ASR↓	TSR↑	ASR↓	TSR↑	ASR↓	TSR↑	ASR↓	TSR↑
TT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
AB	0.0	0.5	0.0	34.0	0.0	55.5	0.0	62.3	0.0	38.1
TEH	0.0	0.3	0.0	48.8	0.0	61.5	0.0	68	0.0	44.6
CRP	14.3	51.5	33.3	50.0	0.0	69.0	0.0	77.7	11.9	62.1
CRP + TEH	0.0	50.5	0.0	55.8	0.0	78.5	0.0	77.7	0.0	63.3

Each strategy offers a different trade-off. Task Termination (TT) is the most stringent approach; it neutralizes threats by terminating any task when an attack is detected. This method is impractical for real-world scenarios because it results in a complete task failure (0% TSR). Action Blocking (AB) is a less severe strategy that blocks a single malicious action but allows subsequent, non-malicious actions to proceed, forcing the agent to replan. Tool Execution Halt (TEH) offers a more granular approach. A single agent "action" can invoke multiple tool calls (some benign), so TEH blocks only the suspicious tool call—not the entire action—letting the agent continue its plan with a "no tool response" error. In contrast, Collaborative Re-planning (CRP) is the least invasive method. Instead of blocking, VeriGuard sends an alert to the agent, which allows it to formulate a new, safer plan. While this significantly boosts the TSR, it doesn't guarantee security, as the agent can still perform unsafe actions (leading to an 11.9% average ASR). Therefore, a hybrid CRP + TEH approach yields the optimal results. This combination leverages the high TSR of CRP with the fine-grained security of TEH, achieving both a near-zero average ASR (0.1%) and the highest average TSR (63.6%).

6 Conclusion

In this work, we introduce VeriGuard, a novel framework designed to substantially enhance the safety and reliability of Large Language Model (LLM) agents. By integrating a verification module that formally checks agent-generated policies and actions against predefined safety specifications, VeriGuard moves beyond reactive, pattern-matching safety measures to a proactive, provably-sound approach. Our experiments demonstrate that this interactive verification loop is highly effective at preventing a wide range of unsafe operations, from prompt injections to unauthorized data access, while maintaining a high degree of task success. The results on benchmarks such as ASB, EICU-AC, and Mind2Web-SC show that VeriGuard not only significantly reduces the attack success rate to near-zero but also offers flexible policy enforcement strategies that can be tailored to different operational needs. VeriGuard provides a robust and essential safeguard, paving the way for the trustworthy deployment of LLM agents in complex and high-stakes real-world environments.

References

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andrew Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Grosse, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. Constitutional ai: Harmlessness from ai feedback, 2022.

Chi-Min Chan, Jianxuan Yu, Weize Chen, Chunyang Jiang, Xinyu Liu, Weijie Shi, Zhiyuan Liu, Wei Xue, and Yike Guo. Agentmonitor: A plug-and-play framework for predictive and secure multi-agent systems, 2024. URL https://arxiv.org/abs/2408.14972.

Zhaorun Chen, Mintong Kang, and Bo Li. Shieldagent: Shielding agents via verifiable safety policy reasoning. *ICML*, 2025.

M. Eilers, M. Schwerhoff, A. J. Summers, and P. Müller. Fifteen years of viper. In Ruzica Piskac and Zvonimir Rakamarić (eds.), *Computer Aided Verification (CAV)*, pp. 107–123, Cham, 2025. Springer Nature Switzerland. doi: 10.1007/978-3-031-98668-0_5. URL https://link.springer.com/chapter/10.1007/978-3-031-98668-0_5.

Marco Eilers and Peter Müller. Nagini: a static verifier for python. In *International Conference on Computer Aided Verification*, pp. 596–603. Springer, 2018.

Deep Ganguli, Liane Lovitt, Jackson Kernion, Amanda Askell, Yuntao Bai, Saurav Kadavath, Ben Mann, Ethan Perez, Nicholas Schiefer, Kamal Ndousse, Andy Jones, Sam Bowman, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Nelson Elhage, Sheer El-Showk, Stanislav Fort, Zac Hatfield-Dodds, Tom Henighan, Danny Hernandez, Tristan Hume, Josh Jacobson, Scott Johnston, Shauna Kravec, Catherine Olsson, Sam Ringer, Eli Tran-Johnson, Dario Amodei, Tom Brown, Nicholas Joseph, Sam McCandlish, Chris Olah, Jared Kaplan, and Jack Clark. Red teaming language models to reduce harms: Methods, scaling behaviors, and lessons learned, 2022. URL https://arxiv.org/abs/2209.07858.

Soroush Ghaffarian, Ruturaj Raval, Gabriele Bavota, and Maliheh Izadi. Can llms verify their own code? a case study in secure web development, 2024.

Boyu Gou, Zanming Huang, Yuting Ning, Yu Gu, Michael Lin, Weijian Qi, Andrei Kopanev, Botao Yu, Bernal Jiménez Gutiérrez, Yiheng Shu, Chan Hee Song, Jiaman Wu, Shijie Chen, Hanane Nour Moussa, Tianshu Zhang, Jian Xie, Yifei Li, Tianci Xue, Zeyi Liao, Kai Zhang, Boyuan Zheng, Zhaowei Cai, Viktor Rozgic, Morteza Ziyadi, Huan Sun, and Yu Su. Mind2web 2: Evaluating agentic search with agent-as-a-judge, 2025.

Significant Gravitas. Auto-gpt: An autonomous gpt-4 experiment. https://github.com/Significant-Gravitas/Auto-GPT, 2023.

Hakan Inan, Kanishka Kandasamy, Sarath Rameshbabu, Moustafa El-Khamy, Suman Purohit, and Srivatsa Ranganath. Llama guard: Llm-based input-output safeguard for human-ai conversations, 2023. URL https://ai.meta.com/research/publications/llama-guard-llm-based-input-output-safeguard-for-human-ai-conversations/.

Neel Jain, Avi Schwarzschild, Yuxin Wen, Gowthami Somepalli, John Kirchenbauer, Ping yeh Chiang, Micah Goldblum, Aniruddha Saha, Jonas Geiping, and Tom Goldstein. Baseline defenses for adversarial attacks against aligned language models, 2023. URL https://arxiv.org/abs/2309.00614.

Zhihan Jin, Hongyi Zhang, Zhiming Zhou, Jiachen Li, Min Gao, and Enhong Chen. Llm-safeguard: A human-in-the-loop framework for tuning safety-guard of llm-based agents, 2024.

Guanting Li, Yifeng Zhang, Zhaohua Chen, Hongwei Wang, Zhaoran Wang, Si-Qing Chen, Yan-Fu Li, Zhuo Tang, Mas ud K. Effendy, An-Tho T. Nguyen, Xiaofei Xie, Meng-Hsun Tsai, and Ting-Chen Chen. Llm-based generation of verifiable computation, 2024.

AI @ Meta Llama Team. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Weidi Luo, Shenghong Dai, Xiaogeng Liu, Suman Banerjee, Huan Sun, Muhao Chen, and Chaowei Xiao. AGrail: A lifelong agent guardrail with effective and adaptive safety detection. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8104–8139, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.399. URL https://aclanthology.org/2025.acl-long.399/.

Justus Mattern, Fatemehsadat Mireshghallah, Zhijing Jin, Bernhard Schoelkopf, Mrinmaya Sachan, and Taylor Berg-Kirkpatrick. Membership inference attacks against language models via neighbourhood comparison. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 11330–11343, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.719. URL https://aclanthology.org/2023.findings-acl.719/.

Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. Automatically correcting large language models: Surveying the landscape of diverse automated correction strategies. *Transactions of the Association for Computational Linguistics*, 12:484–506, 2024. doi: 10.1162/tacl_a_00660. URL https://aclanthology.org/2024.tacl-1.27/.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 126544–126565. Curran Associates, Inc., 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/e4c61f578ff07830f5c37378dd3ecb0d-Paper-Conference.pdf.

Traian Rebedea, Razvan Dinu, Makesh Narsimhan Sreedhar, Christopher Parisien, and Jonathan Cohen. NeMo guardrails: A toolkit for controllable and safe LLM applications with programmable rails. In Yansong Feng and Els Lefever (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 431–445, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-demo.40. URL https://aclanthology.org/2023.emnlp-demo.40.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Tsvigun, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023a.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023b.

Sander V Schulhoff, Jeremy Pinto, Anaum Khan, Louis-FranÃ§ois Bouchard, Chenglei Si, Jordan Lee Boyd-Graber, Svetlina Anati, Valen Tagliabue, Anson Liu Kost, and

Christopher R Carnahan. Ignore this title and hackaprompt: Exposing systemic vulnerabilities of llms through a global prompt hacking competition. In *Empirical Methods in Natural Language Processing*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhi-Yuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. A survey on large language model based autonomous agents, 2023.

Alexander Wei, Nika Haghtalab, and Jacob Steinhardt. Jailbroken: How does llm safety training fail? *NeurIPS*, 2023.

Zhengyao Xi, Wenxiang Chen, Xin Guo, Wei He, Yi Ding, Bowei Hong, Ming Zhang, Jun Wang, Sen Jin, Enyi Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

Zhen Xiang, Linzhi Zheng, Yanjie Li, Junyuan Hong, Qinbin Li, Han Xie, Jiawei Zhang, Zidi Xiong, Chulin Xie, Carl Yang, Dawn Song, and Bo Li. Guardagent: Safeguard llm agents by a guard agent via knowledge-enabled reasoning, 2025.

Xilie Xu, Keyi Kong, Ning Liu, Lizhen Cui, Di Wang, Jingfeng Zhang, and Mohan Kankanhalli. An llm can fool itself: A prompt-based adversarial attack, 2023. URL https://arxiv.org/abs/2310.13345.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.

Hanrong Zhang, Jingyuan Huang, Kai Mei, Yifei Yao, Zhenting Wang, Chenlu Zhan, Hongwei Wang, and Yongfeng Zhang. Agent security bench (asb): Formalizing and benchmarking attacks and defenses in llm-based agents. In *The Thirteenth International Conference on Learning Representations*, 2025.

Yicong Zhao, Shisong Chen, Jiacheng Zhang, and Zhixu Li. Recode: Improving llm-based code repair with fine-grained retrieval-augmented generation, 2025. URL https://arxiv.org/abs/2509.02330.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

A Experiment Details

This section details the implementation VeriGuard with agent systems mentioned in Section 4.

A.1 Prompts Used To Generate the Policy

Prompt: Policy Code Generation

You are an expert AI security agent. Your primary function is to generate a Python security policy function based on a high-level user request.

CONTEXT

You are part of a system that moderates a target agent's actions at runtime. This system works as follows:
1. A user gives a natural language security request (e.g., "The agent cannot send emails to non-company addresses").
2. You receive this request and generate a specific Python `policy` function to enforce it.
3. At runtime, before the target agent acts, its inputs and outputs are intercepted.

4. Parameters are extracted from this runtime data using the `read_parameters_with_LLM` tool.
5. The extracted parameters are passed to your generated `policy` function, which decides whether to allow or deny the action.
Runtime Data:
- `agent_information: str` # The text prompt input given to the agent, the agent output, and any available context.
Parameter Extraction Tool: The system uses the `read_parameters_with_LLM` tool to extract relevant arguments from the `agent_information`.
- **Tool Signature**: `read_parameters_with_LLM(text: str, list_of_arguments_to_read: dict[str, type]) -> dict`
- **Example**: Given the text *"Send an email to alex@example.com and Dona Jhonson with the subject 'Update'"* and `list_of_arguments_to_read={{"recipients": list, "subject": str}}`, the tool would return `{{"recipients": ["alex@example.com", "dona@example.com"], "subject": "Update"}}`.

YOUR TASK

Given a agent's security `request` and the target `agent_specification`, you must generate two outputs: a list of parameters to extract, and the Python code for the policy function.

OUTPUT REQUIREMENTS

Your response must contain exactly two parts in the following order:
1. Arguments to Extract
A Python dictionary named `list_of_arguments_to_read` that contains all the string arguments needed from the runtime data to evaluate the policy. Follow this exact format:

```
```python
 list_of_arguments_to_read = {{"argument1": type1, "argument2":
    type2, "argument3": type3}}
```
```

2. Policy Function Header

```
```python
def policy(arguments: dict) -> tuple[bool, str]:
```
```

3. Constant variables
You can define global variables needed to execute the function.

```
```python
variable1 = "value1"
variable2 = ["value2"]
```
```

4. Policy Function Code
A Python function that implements the user's security policy. You can create auxiliary functions as well, but the policy function must be named exactly `policy`.
The function must adhere to these rules:
- It must be named exactly `policy`.
- It must accept one argument: `arguments: dict`.
- It must return a tuple: `(bool, str)`. The `bool` is the decision: `True` if the action is **allowed**, `False` if it is **denied**.
- It must include a clear docstring explaining the policy.
- It must not use any external libraries.

Note: Do not use python string functions to get parameters or arguments. Use the `read_parameters_with_LLM` tool instead.
Follow this exact format:

````
```python
def policy(arguments: dict) -> tuple[bool, str]:
    """A brief, one-sentence description of what this policy
    enforces."""
    # Your policy logic here
    # ...
    if condition_is_met:
        return True, "Reason for allowing."
    else:
        return False, "Reason for denial."
```
````

—
Target Agent Specification:
{agent_specification}
User Request
{request}
Requirements # If available
{requirements}
Assumptions # If available
{assumptions}

Prompt: Constraint Generation

You are a senior **Security Systems Architect** with a deep understanding of **Natural Language Understanding (NLU)** and formal logic. Your expertise lies in translating complex, human-written security policies into clear, unambiguous, and verifiable logical rules.

Objective

Your task is to analyze a given natural language security policy, an agent's description, and a set of defined variables/constants to generate a set of discrete, testable logical statements. These statements should represent the core requirements of the policy and can be used to programmatically verify the agent's behavior.

Instructions

1. **Analyze the Inputs:** Carefully read the provided security policy, the agent's description, and the list of variables and constants. Identify key actors, actions, conditions, and the specific capabilities of the agent that are relevant to the policy.
2. **Generate Logical Statements:** For each policy rule, create one or more logical statements. * Each statement should be a complete, verifiable condition. * Use a clear **IF-THEN** or similar logical structure. * Focus on verifiable conditions (e.g., "The agent's state is 'idle'") and outcomes (e.g., "The agent must not respond to external commands"). * Ensure the statements cover both "positive" (allowed) and "negative" (denied) cases, taking into account the agent's specific functions.
3. **Provide a Rationale:** After the list, briefly explain the logical structure and why these statements effectively capture the policy's intent and apply to the described agent.
4. **Output Format:**
Verifiable Logical Statements:
(Give the statements in json format as:)

````
    ```json
    [{"statement": ...}},
````
````

```
    {{"statement": ...}}
    ...
    ]```
```

Rationale:
(A brief explanation of the logical framework and how it aligns with the policy, the agent's capabilities, and the provided variables.)

Input:

Security Policy Specification:
{security policy}
Agent Specification:
{agents_description}
Variables and Constants:
{variables_and_constants}
Requirements # If available
{requirements}
Assumptions # If available
{assumptions}

Output:

Verifiable Logical Statements:

A.2 Prompts Used To Enforce the Policy

Prompt: Argument Extraction

You are an expert Natural Language Understanding (NLU) engine. Your task is to extract structured information from a given text based on a specification and return it as a single Python dictionary.

YOUR TASK

You will be provided with a block of input text, a Python function for context, and a specification of the parameters to extract. You must carefully read the **entire** input text, find the values for each required parameter, and format them into a dictionary.
Instructions:
- Synthesize information from all parts of the input text to fill the parameters.
- Ensure the data types in the output dictionary match the specification (e.g., `list` for `list[str]`).
- Your output at the end should include your reasoning and a valid Python dictionary.
—–

EXAMPLE

GIVEN THIS CONTEXT:
Python Function:

```
```python
def send_email(arguments: dict) -> str:
    """Send an email to the specified recipients with the given
    subject."""
    # send email logic here
    # ...
    return "Email sent successfully."
```
```

Agent Information:
Input: Send an email to alex@example.com and Dona Jhonson with the subject 'Update'

Output: Step 1: Using retreive_email("Dona Jhonson") we get the email "dona@example.com"

AND THIS PARAMETER SPECIFICATION:
`{{"recipients": "list[str]", "subject": "str"}}`

PRODUCE THIS OUTPUT:
Reasoning
From the "Agent Input", I found one recipient, "alex@example.com", and the subject, "Updates".
From "Agent Output" Step 1, I found the email for "Dona Jhonson" is "dng@example.com". I will combine these into the ‘recipients‘ list.
Python dictionary

```
```python
{"recipients": ["alex@example.com", "dona@example.com"], "
    subject": "Update"}
```
```

—–

YOUR TURN

GIVEN THIS CONTEXT:
Python Function:
`{function}`
Input Text:
`{text}`

AND THIS PARAMETER SPECIFICATION:
`{parameters}`

PRODUCE THIS OUTPUT:

A.3 Prompts Used for Validation

Prompt: Validation Analysis

You are an expert **Natural Language Understanding (NLU)** and **logic engine**. Your primary function to verify logical statements.

YOUR TASK

Given a user’s security specification and statements, you must analyze the specification in detail and then check if the logical statements is valid or needs correction.
1. Check if the user specification has ambiguity, needs clarification, for example co-references.
2. Check pre-assumptions for the statements. Focus on the specification.
4. Find contra examples.
5. Find any logical error in the statements.
Output: After your analysis list all the points that require clarification or correction.
—
User Specification
`{user_specification}`
Logical Statements
`{statements}`

Prompt: Validation Disambiguation

You are an expert in **System Requirements**, **Security Policy**, and **Logical Deduction**. Your primary function is to act as an arbiter to resolve ambiguities identified in a system analysis. You must review a user's security goals, the agent's capabilities, and the provided analysis to establish a definitive, clear, and reasonable set of system requirements and assumptions.

YOUR TASK

You are given a high-level `user_specification`, the technical `agent_specification`, and an `analysis` that identifies points of ambiguity, conflict, or missing details.
Your task is to:
1. Carefully examine each point raised in the `analysis`.
2. Use the `user_specification` as the primary source of intent and the `agent_specification` as the context for technical constraints.
3. For each point of ambiguity, make a clear and logical **decision** to finalize the requirement or assumption.
4. Compile these decisions, along with any original unambiguous requirements, into a single, comprehensive list of detailed requirements.

OUTPUT FORMAT

Your response must contain two parts:
Part 1: Decisions on Ambiguities
For each point from the analysis, provide your decision in the following structured format:
1. [Title of the Point/Ambiguity]
- Decision: [State your clear and final decision on the requirement or assumption.]
- Justification: [Briefly explain *why* this decision is the most reasonable, referencing the user/agent specifications as needed.]
2. [Title of the Next Point/Ambiguity]
- Decision: [...]
- Justification: [...]
Part 2: Finalized Detailed Requirements List
After addressing all ambiguities, compile a complete and final list of all detailed requirements (combining the original, clear requirements with your new decisions).
1. [Detailed Requirement 1]
2. [Detailed Requirement 2]
3. [Detailed Requirement 3]
—

INPUTS

User Specification
`{user_specification}`
Agent Specification
`{agent_specification}`
Analysis of Ambiguities
`{analysis}`

A.4 Prompts Used for Code Testing

Prompt: Test Case Generation

You are an expert at writing Pytest functions. Your task is to generate complete and effective test cases for a given Python function, adhering to best practices.

YOUR TASK

Generate Pytest functions within a single Python code block. The tests should be comprehensive, covering a wide range of scenarios including:
- **Happy Path:** Standard, valid inputs.
- **Edge Cases:** Boundary conditions (e.g., empty strings, zero, negative numbers).
- **Error Handling:** Cases that should raise specific exceptions.
Use the following format for your output:

````
```python
# your generated test code here
```
````

User Request:
{user_request}
Requirements
{requirements}
Assumptions
{assumptions}
Python function to test:
{function_to_test}
Test cases:

Prompt: Policy Code Correction

You are an expert Python developer and debugger. Your task is to analyze a Python function and its corresponding pytest error message, identify the bug, and provide the corrected code.
Python Function to Correct
{function_to_test}
Pytest Error Message
{error_message}

Your Task

Analyze the function and the error message to find the source of the error.
Explain the bug clearly and concisely.
Provide the complete, corrected Python function.
Response Format
Bug Explanation
(Describe the bug and the reason for the error here.)
Corrected Function

````
```python
# Your corrected Python code here.
```
````

A.5 Prompts Used for Verification

Prompt: Code Generation for Verification

You are an expert in **formal methods** and **software verification**, specializing in Python. Your primary skill is translating requirements into precise **Nagini pre- and post-condition contracts**.

Objective:

Your task is to augment a given Python function with Nagini contracts (‘Requires‘ and ‘Ensures‘) based on a set of logical statements. You must ensure the generated code is syntactically correct and accurately reflects the logic of the provided statements.

Instructions

1. **Analyze the Inputs:** Carefully review the provided Python function and the list of requirements given as logical statements.
2. **Translate Policies to Nagini:** For each logical statement, formulate the equivalent Nagini ‘Requires‘ (pre-conditions) or ‘Ensures‘ (post-conditions).
3. **Adhere to Grammar:** Strictly follow the provided Nagini grammar and refer to the examples for correct syntax and structure.
4. **Integrate and Output:** Embed the generated Nagini contracts directly into the Python function.

Inputs

Python Function:
`{python_function_code}`
Requirements:
`{list_of_logical_statements}`
Nagini Grammar Reference:
`{grammar}`
Nagini Examples:
`{examples}`

Output Format

Provide the complete Python code for the function, including the newly added Nagini decorators, inside a single Python code block.