

Original Research

Received 09 March 2026

Revised 12 April 2026

Accepted 25 May 2026

Natural Resources for Human Health 2026, Vol. 6 (17s): 778–801

KEYWORDS:

LLM security, autonomous agents, prompt injection, retrieval-augmented generation, semantic drift, zero-trust architecture, runtime guardrails, multi-turn jailbreak

Semantic Drift in Autonomous Agents: Detecting Runtime Goal Mutation in Multi-Agent LLM Systems

Shubhra Mukherjee Mathur¹, Naga Sai Khyati Paruchuri², Priyanka Abhang³, Sumedha Umakant Ayachit⁴, Swati Mohan Patil⁵, Manasa Manaswini Paruchuri⁶

^{1,6} School of Computing, MIT Art Design Technology University, Pune (mukherjeeshubhra@gmail.com), (sumedha.ayachit@mituniversity.edu.in) (manasaparuchuri100@gmail.com),

² Mini Orange, Pune (pnskhyati@gmail.com)

³ G H Raisoni International Skill Tech University, Pune (priyanka.abhang@gmail.com)

ABSTRACT: Autonomous large language model (LLM) agents extend the attack surface beyond classical input-output filtering. Tool outputs, retrieval-augmented generation (RAG) chunks, inter-agent messages, and persistent memory constitute internal trust boundaries that adversaries exploit through indirect prompt injection, RAG poisoning, semantic goal drift, and multi-turn crescendo jailbreaks. We present GuardAgent Runtime, a framework-agnostic middleware that interposes on agent execution, enforces zero-trust inspection of all untrusted data flows, and maintains runtime intent continuity relative to the user's authorized objective. GuardAgent composes pattern-based injection detection, embedding-based semantic drift and intent verification (Shadow Critic), multi-layer RAG inspection, and a temporally decaying risk accumulation engine with crescendo trend detection. We formalize detection scores and policy mapping, describe an open-source implementation with LangGraph/CrewAI/AutoGen adapters, and evaluate on a 20-scenario adversarial benchmark plus a browser pre-submit extension. With isolated per-scenario evaluation, GuardAgent achieves 90.0% F1 (90.0% precision, 90.0% recall), 100% accuracy on RAG poisoning and benign corpora, and mean interception latency under 4 ms excluding cold-start embedding load.

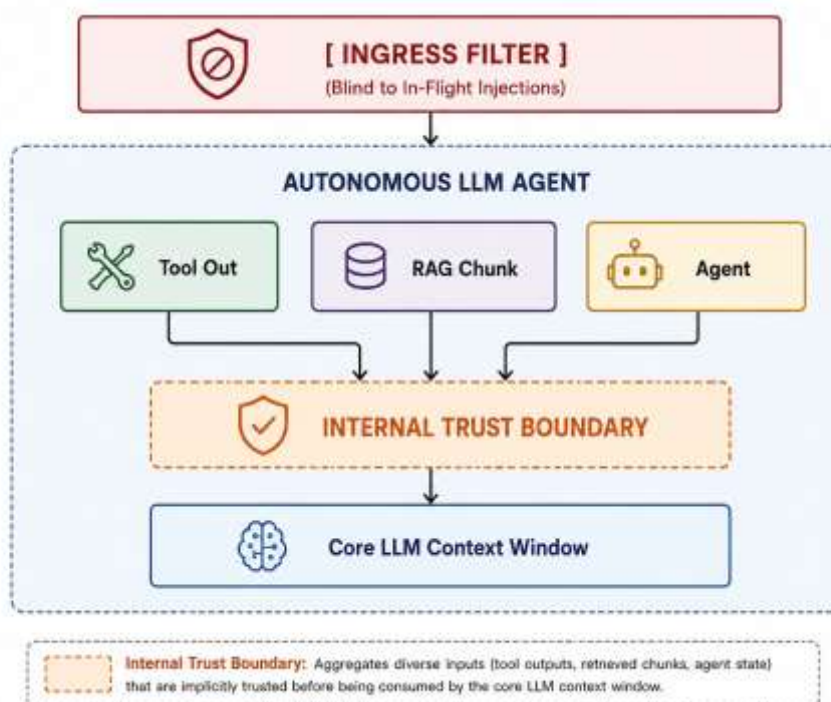
1. INTRODUCTION

The paradigm shift from passive Large Language Model (LLM) text generators to autonomous, goal-driven LLM agents represents a major leap in artificial intelligence capabilities [32]. By coupling core reasoning engines with external tool execution environments, retrieval-augmented generation (RAG) pipelines [21], multi-agent orchestration frameworks, and long-term memory systems, autonomous agents can independently plan, execute, and iterate on complex workflows. However, this transition from isolated input-output mapping to stateful, dynamic software execution drastically expands the systems-level attack surface beyond classical input-output filtering.

Within traditional large language model structures, protection limits remain in a fixed position because protection limits focus mainly on the first user input and the end model answer. The system is changed completely by autonomous agents because autonomous agents allow every middle piece of data to enter the model environment again. Since every piece of information that is produced during the process can return to the model context and change the model way of acting, every location where outside information is received is seen as a boundary that is not confirmed. Malicious individuals use smart and hidden paths to reach these internal trust limits because these malicious individuals want to bypass the safety systems. Instead of performing

a direct attack on the autonomous agents, a person with bad intent might corrupt a public website which starts an indirect prompt injection [2] or quietly damage a RAG storage place to create a change in the meaning of the goal. This change makes the autonomous agents leave the original path that was allowed for the autonomous agents. It has been observed that long-term memory allows for many-step attack plans to function well, including crescendo jailbreaks [6] in these crescendo jailbreaks [6], harmful goals are increased slowly and spread across steps that look safe so that the simple safety rules are avoided.

Figure 1 : The expanded internal attack surface of autonomous LLM agents bypassing traditional ingress perimeters.



The Open Web Application Security Project (OWASP) places prompt injection and data leakage at the top risks for LLM applications [1] [33]. This consensus is, even though our analysis demonstrates an abundance of existing production deployments which exclusively use perimeter-focused ingress/egress filters that ignore critical in-flight corruption. Existing LLM defense mechanisms are fundamentally incapable for this agentic environment ([33] [34]), as neural networks may not be inherently adversarially aligned [25]. Input-output firewalls are stateless, oblivious to intermediate runtime flows. Hard-coded wrapper prompts reduce model reasoning and suffer when adversarial payloads are injected deep into tool payloads. Even more importantly, existing guardrails are context unaware; they examine data flows in an absolute isolation, completely oblivious of whether an agent's current execution path still adheres to the user's initial, authorized, mission objective.

To bridge this gap, humans present GuardAgentRuntime, which is a software layer that works across different frameworks to stand between agent execution directly. Because GuardAgentRuntime follows a zero-trust model, GuardAgentRuntime treats every piece of internal information like tool produced results, RAG retrievals, and memory—as if the information pieces are naturally untrusted. Protection measures are applied by the runtime without changing the weights of the LLM or stopping the flexibility of the agent.

This structural design includes a multi-layered defense pipeline. Experts claim [15] that the ShadowCritic engine is used to check if the internal path of an agent is moving away from its goal. To stop bad use that grows over time, GuardAgentRuntime includes a risk engine that calculates growing danger to catch small attacks before they break rules.

1.1 Our Contributions

The primary contributions of this work are as follows:

- **Threat Modeling of Internal Trust Boundaries:** We formalize the attack surface of autonomous agents, providing explicit interception points and categorizing vulnerabilities across tools, RAG retrieval chunks, memory stores, and multi-agent communications.
- **The GuardAgent Framework:** We design and implement a framework-agnostic, zero-trust middleware architecture featuring multi-layer RAG inspection and a stateful, temporally decaying risk accumulation engine for crescendo trend detection.
- **Formalized Multi-Layer Policy Mapping:** We formalize dynamic scoring systems for pattern-based injection, semantic goal drift, intent verification, and RAG relevance coupled with strict policy enforcement.
- **Production Adapters & Evaluation:** We present an open-source implementation with production-ready adapters for prominent agent frameworks (LangGraph, CrewAI, AutoGen) [12] [13] [14] a modular plug-in extension API, and a *Chrome Prompt Shield* browser extension for pre-submit scanning. We report empirical benchmark metrics evaluated across a labeled 20-scenario adversarial attack suite, demonstrating a **90.0% F_1 -score** (90.0% precision, 90.0% recall) along with **100%** accuracy on RAG poisoning and benign corpora, while explicitly discussing observed failure modes.

2. RELATED WORK

2.1 Prompt Injection Defenses

Contemporary prompt injection defenses primarily rely on edge-focused pattern matching and classifier guardrails. Modern ways to protect against prompt injection depend mostly on checking patterns at the edges. It has been observed that systems like Rebuff and LLMGuard [9] [10] look at user prompts to stop bad data before the data reaches the orchestration layer. Methods like StruQ [11] and general taxonomies of injection vectors [3] [18] [23] are used to keep instructions separate from data by using strict query rules. While these methods work for one-turn attacks, these tools are not often placed for internal paths, which leaves tool returns and other information streams open to attacks.

2.2 Orchestration Frameworks and Safety Ecosystems

Creating autonomous agents today uses frameworks like LangGraph, CrewAI, AutoGen, and the OpenAI Agents SDK [12] [13] [14] [24] to handle communication and planning. These systems are made for performance rather than keeping information pieces safe. Many people believe [34] that these frameworks, even when paired with vendor-specific guardrails [26] [27], give the job of protection to the application itself. If a tool or memory is broken, the framework sends the broken information forward. GuardAgentRuntime fixes this structural gap because GuardAgentRuntime wraps around these frameworks with simple adapters.

2.3 Semantic Vector Embeddings for Safety

Using thick vector spaces for matching language is a common method. Computational structures like Sentence-BERT [5] and classical information retrieval techniques [19] make it possible to check meanings quickly. It has been observed that keeping execution separate using two computational structures allows one structure to watch the other [15]. GuardAgentRuntime uses this idea with the ShadowCritic engine. Fast calculations are done by the system [5] to verify what the agent wants to do without making the system slow.

2.4 Stateful Multi-Turn Attacks

New information about threats shows that smart attacks and black-box jailbreaking techniques [17] can get past firewalls by slowly changing what the computational structure wants to do [6] Because the malicious information is spread over many small steps, the malicious individual does not set off protection alarms. Experts claim [6] that protecting agent environments needs a way to track things over time. This danger is fought by GuardAgentRuntime through a risk engine that finds when a threat is growing.

2.5 Zero-Trust Architecture for AI Systems

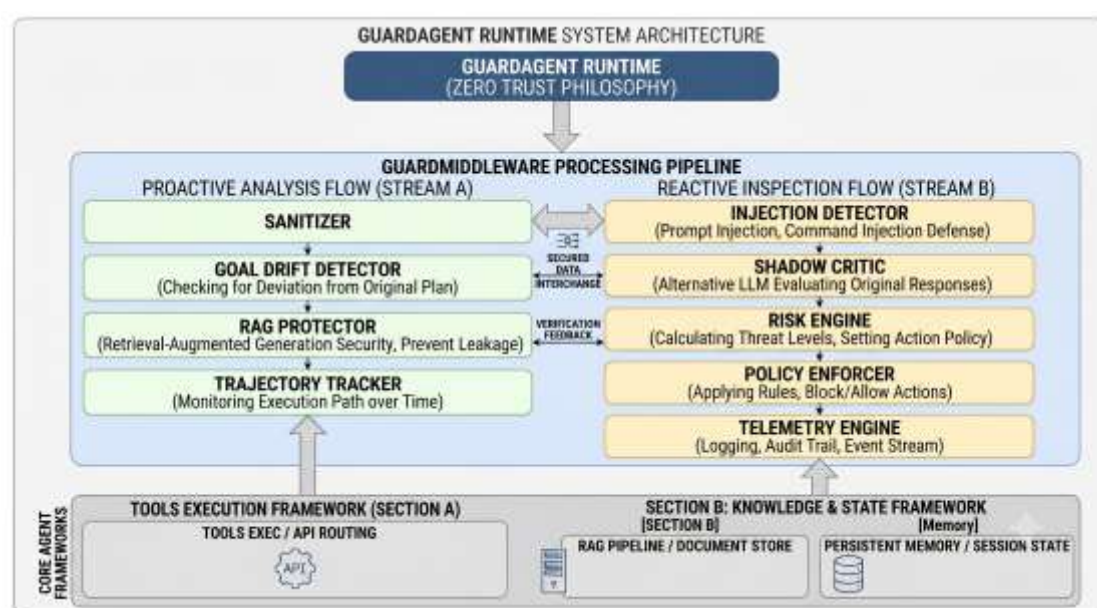
The main rule of the NIST SP 800-207 standard [8] and evolving AGI safety levels [16] [28] say that one must never trust and always check every path where services talk to each other. It has been observed [8] that networks and address mapping protocols

[32] must be treated as dangerous places. While classically constrained to network infrastructure, we port this principle directly to the internal data-flow architecture of LLM agents. GuardAgent treats every internal transition point—including tool returns, RAG fragments, sub-agent syncs, and memory updates—as an untrusted boundary requiring active sanitization, inspection, and policy verification.

3. SYSTEM ARCHITECTURE

We realize our zero-trust philosophy through the implementation of the *GuardAgent Runtime*, structured as a decoupled GuardMiddleware processing pipeline. The framework intercepts data exchanges between the core orchestration abstractions (e.g., LangGraph, CrewAI, AutoGen) and downstream operations through transparent system adapters.

Figure 2: GuardAgent Runtime layered architecture showcasing the sequential decoupling of the execution pipeline from framework primitives.



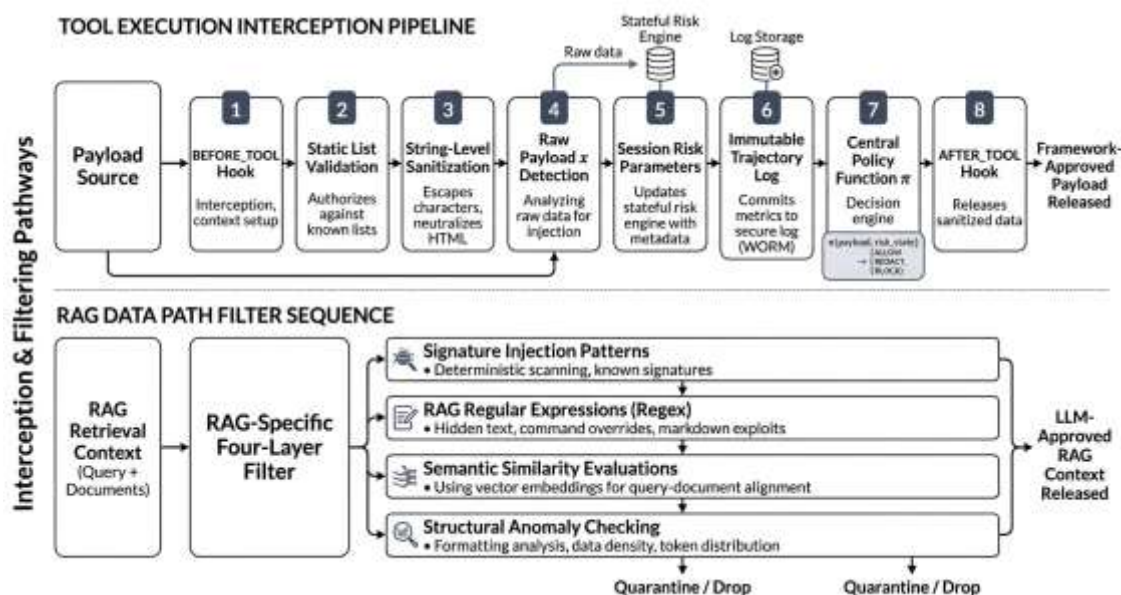
The GuardMiddleware architecture arranges its protection layer through a structured sub-component system, it is shown in Fig. 2:

1. **Ingestion and Signature Filtering:** The job of the *Sanitizer* and *Injection Detector* is to manage the starting low-overhead stings verification. Once they get the clean raw string as an initial format they apply regex rules to know and guess the malicious payloads from the output given by the tool.
2. **Semantic & Alignment Core:** The data that actually passed through the first layer is now sent to *Goal Drift* evaluator and the *Shadow Critic*. In the *Shadow Critic* layer it calculated how close the meaning is in real time using Sentence-BERT vector spaces [5]. The *Shadow Critic* uses this calculation to validate that every step matches the real intention of the user. Experts claim [5] that mathematical alignment helps stop errors during the middle parts of the process.
3. **Contextual Storage & Stateful Risk Assessment:** *RAG Protector* examines how much the pieces of information relate to each other so that database manipulation can be found [4] [21]. While this occurs, the *Risk Engine* follows the history of every time the user speaks to the machine. A mathematical penalty that decreases over time is applied by the system to mark attacks that happen over many turns [6] Many people believe [6] that tracking history is the best way to see hidden threats.
4. **Enforcement and Observability:** The *Trajectory Tracker* keeps a record of all the things the agent does. At the same time, the *Policy Enforcer* compares the danger scores that happen in the moment against the security rules that were set before. This comparison is done so that the system can stop actions that are not allowed. The *Telemetry Engine* sends

out organized data reports that large organizations use for watching their systems. It has been observed [8] that structured logs are helpful for high-level monitoring.

3.1 Interception Pipeline

Figure 3: Architectural overview of the security pipelines: The 8-step Tool Execution Interception Pipeline (top) processing payload x at turn t , and the parallel 4-layer RAG Data Path Filter Sequence (bottom) for mitigating retrieval-based exploits.



When a tool gives back an external payload x during the time t of the conversation, the flow of data is caught by the execution pipeline. Execution pipeline operates the payload x successively through eight different parts of the work. Validating the data at the beginning helps identify issues early and reduces the risk of errors later in the process.

1. **Trigger the BEFORE_TOOL Lifecycle Hook:** The hook is resembling to the first guard at the gate, it catches the raw payload x right after the tool has completed its job. The hook stops the main thread of the application so that the time for turn t can be recorded. Experts claim [30] that the hook prepares the whole path for handling the data.
2. **Validate Invocation Against Static Tool Allow/Block Lists:** The system looks at the name of the tool and the shape of the output to check them against lists of security rules that cannot be changed. Authorization for the current situation is verified by allow-lists. At the same time, block-lists throw away payloads that come from tools that are old or broken. It is believed [1] that static lists provide strong defense.
3. **Perform String-Level Sanitization:** This part of the work performs changes on the textual payload. Harmful characters are removed and dangerous HTML or XML tags are made harmless by the system. For example, `<script>` is changed into normal text. It has been observed [10] that cleaning the data stops anomalies in the coding.
4. **Execute Pattern-Based Injection Detection:** Pattern looking and heuristic searching happen directly on the raw, unsanitized payload x that was kept from the first step. Analyzing the first state of the data is very important because it stops bad information from hiding behind changes. Many people believe [9] that checking the original state is the only way to find hidden attacks.
5. **Pass Metadata to Update Stateful Session Risk Parameters:** Security metadata that comes from the steps before is put into a risk engine that remembers past events. The risk engine updates the cumulative session risk score. Because the risk engine works this way, the system can stop or end sessions when the danger becomes too high over many turns. Experts claim [6] that behavioral flags are essential for safety.

6. **Commit Turn Metrics to the Immutable Trajectory Log:** To maintain strict auditability, compliance, and forensics, the turn metrics, cryptographic hashes of the payload, and execution states for turn t are appended to an immutable, write-once-read-many (WORM) trajectory log.
7. **Apply the Central Policy Function π :** The pipeline evaluates the sanitized data and stateful session parameters against the core security policy function π . This mathematical mapping determines the final allowance state:

$$\pi(\text{payload}, \text{risk_state}) \rightarrow \{\text{ALLOW}, \text{REDACT}, \text{BLOCK}\}$$

Depending on the output, the payload is passed through, subjected to PII redaction, or discarded entirely.

8. **Fire the Closing AFTER_TOOL Hook:** The complete payload is given by the final hook, now this shows that it is according to the core application framework and it is complete and approved. It removes all the short-term memory buffers, completes the turn processing metrics, and tells the LLM execution loop to proceed further safely.

To stop the retrieval-based exploits, RAD data paths skips the standard tool process and goes through a different, specially made four layer filter system:

- **Signature Injection Patterns:** A deterministic, low-latency scanning layer that inspects retrieved document chunks for known static attack signatures, common jailbreak templates, and explicit adversarial instructions.
- **RAG-Specific Regular Expressions:** When someone tries to modify the system instructions or steal data, we have specific filters with specialized regular expressions also known as regex rules to detect. The job of these filters is to look for hidden tricks in the data or empty spaces that are usually used to hide malicious text from the normal human. But it is observed [10] that these filters catch a limited threats.
- **Query-Document Semantic Similarity Evaluations:** This layer finds the distance between what the user asked for and the context that is found using mathematics. The term for this is *Semantic distance*, in this we measure the Semantic distance, which tells the system if the answer is correct or not.
- **Structural Anomaly Checking:** The structural layout and the incoming data format checking is the task of the final layer. This finds the structural mutations, unnatural token distributions, hidden code blocks masked as prose, or unexpected high-density data payloads (e.g., base64 strings embedded in natural text).

4. THREAT MODEL AND ASSUMPTIONS

To analyze the security guarantees of our framework, we created a formal threat model showcasing all the capabilities of malicious actor, the execution context, and the trusted computing base (TCB). We examined an agent executing actions to satisfy an authorized user's main goal g .

4.1 Adversarial Capabilities

We assumed an unauthorized, external adversary $\mathcal{A}$ whose main goal is to hijack the agent's execution flow, extract sensitive session data, or generate unauthorized state transitions (e.g., fraudulent API calls). The adversary $\mathcal{A}$ does not have direct code execution authorities on the host system, but they can change the agent's reasoning loop by injecting malicious payloads into downstream data dependencies. Particularly, we assume $\mathcal{A}$ can change the following spaces:

1. **Manipulated Tool/API Return Values:** The attacker can control or intercept communications with external services. When the agent calls an external API, there is a chance that $\mathcal{A}$ returned payload x is exchanged by an attacker at time t . These unauthorized party may hide secret commands or harmful control sequences inside the data. Previous work has shown external services as a significant attack vector for indirect prompt injection attacks [2].
2. **Poisoned RAG Corpus Chunks:** The attacker can inject structured or unstructured information into external knowledge bases, web pages, or corporate repositories/ database. During the query time, these attacked documents are retrieved by the Retrieval-Augmented Generation (RAG) system, these add untrusted contexts into the agent's context window.

3. **Inter-Agent and Memory Exploitation:** In multi-agent systems, $\mathcal{A}$ can attack a secondary, intermediate agent. As a result, the attacker can change the inter-agent communication channels and write attacker's payloads to the shared memory cache or long-term trajectory logs read by the main agent.
4. **User-Facing Prompts (Optional Scope):** Only in certain environments, not for every AI application like browser extension or a shared workspace application, we assume a case where $\mathcal{A}$ can change user inputs using Cross-Site Scripting (XSS) or UI deception, sending attackers' input directly into the main prompt stream.

4.2 Trust Assumptions and Trusted Computing Base

Our architecture is built such that it follows certain trusted rules:

- **Integrity of the Middleware:** We assume that the compiled GuardAgent system software is not modified or infected. All the operating systems, libraries, and execution environments are secure, and the security rules (e.g., what tools are allowed or blocked) π cannot be changed by the attacker. The attacker cannot change the security rules and also cannot change the execution pipeline hooks (BEFORE_TOOL and AFTER_TOOL), or delete records within the immutable trajectory log.
- **Data Infrastructure Status:** We assume an *honest-but-curious* model for core tool hosts and database administrators. While they faithfully execute queries and route data, the actual document content or third-party web pages stored within those corpora must be treated as inherently compromised.

4.3 Out of Scope

The following attack vectors are orthogonal to the design of this runtime execution pipeline and are explicitly considered out of scope for this work:

- **Model Weight Poisoning:** Traditional adversarial attacks involving backdooring or poisoning the foundation model's weights during pre-training or fine-tuning phases.
- **Side-Channel and Hardware Attacks:** Hardware-level exploits (e.g., Rowhammer, speculative execution side-channels) aimed at extracting weights or context tokens from GPU/CPU memory clusters.
- **Denial-of-Service (DoS):** Direct resource exhaustion attacks targeting the availability of the middleware framework, API rate-limiting structures, or basic network availability.

5. MATHEMATICAL FORMULATIONS AND SPACES

This section formalizes the universal mathematical spaces, operators, and quantitative models driving the GuardAgent Runtime verification engine. All the configurations are evaluated as a particular projection of all these large mathematical rules. Experts say the security is strong because it is based on a mathematical foundation [5].

5.1 Notation and Underlying Vector Spaces

Let $\mathcal{T}$ be the universal set of all bounded text strings. A string $x \in \mathcal{T}$ is defined as a finite sequence of Unicode code points. The text is projected on to a continuous vector space of dimension $d \in \mathbb{N}$ (where $d = 384$ for the baseline configuration) through geometric operations.

Table 1 presents the global mathematical notation used through this framework.

Table 1 : Mathematical Notation Matrix

Symbol	Operational Semantics
$\mathcal{T}$	Set of valid Unicode text inputs
d	Number of embedding dimensions ($d \in \mathbb{N}$)
$\mathbb{R}^d$	Continuous semantic vector space

Symbol	Operational Semantics
$\mathbb{S}^{d-1}$	Unit-normalized embedding space after- ℓ_2 normalization
a_t, x, q, c	Actions, queries, tool outputs, and RAG chunks
g	Authorized user goal
t	Conversation turn number ($t \in \{1, \dots, T\}$)
τ	Fixed decision thresholds $[0,1]$
$\mathbf{1}[\cdot]$	Binary indicator function

5.2 Distributed Representations and Embedding Geometry

5.2.1 Parametric Encoder Mapping

A frozen parametric nonlinear function maps textual strings to dense vector spaces:

$$f_\theta: \mathcal{T} \rightarrow \mathbb{R}^d, \quad \mathbf{e}_x = f_\theta(x)$$

where θ are fixed weights learned with contrastive loss objectives.

5.2.2 Projection onto the Unit Hypersphere

The raw vectors are normalized to unit ℓ to give scale invariance in the geometric comparison ℓ_2 -normalization projection onto the unit hypersphere $\mathbb{S}^{d-1}$:

$$\hat{\mathbf{e}}_x = \frac{\mathbf{e}_x}{\|\mathbf{e}_x\|_2} = \frac{f_\theta(x)}{\sqrt{\sum_{i=1}^d (f_\theta(x))_i^2}}$$

This bounds $\|\hat{\mathbf{e}}_x\|_2 = 1$, reducing downstream cosine alignments to simple dot products.

5.2.3 Bounded Document Truncation

Prior to representation, high-volume retrieval documents undergo a structural truncation operator $\pi_k: \mathcal{T} \rightarrow \mathcal{T}_{\leq k}$ bounded at $k = 512$ characters:

$$\pi_{512}(x) = x_{1 \dots \min(|x|, 512)}$$

5.2.4 Geometric Cosine Formulations

For any unnormalized representations $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d \setminus \{\mathbf{0}\}$, semantic cosine alignment is formulated as:

$$\cos(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a}^\top \mathbf{b}}{\|\mathbf{a}\|_2 \|\mathbf{b}\|_2} \in [-1, 1]$$

Utilizing hyperspherical unit projections, this reduces uniformly to the inner product:

$$\text{sim}(\mathbf{a}, \mathbf{b}) = \hat{\mathbf{a}}^\top \hat{\mathbf{b}} = \sum_{i=1}^d \hat{a}_i \hat{b}_i$$

For a sequence trace yielding an embedding array, the complete intra-trajectory relationship is captured by the Gram matrix $\mathbf{S}$ derived from the row-normalized design matrix $\tilde{\mathbf{E}} \in \mathbb{R}^{n \times d}$:

$$\mathbf{S} = \tilde{\mathbf{E}} \tilde{\mathbf{E}}^\top \in \mathbb{R}^{n \times n}, \quad S_{ij} = \text{sim}(\mathbf{e}_i, \mathbf{e}_j)$$

5.3 Formal Detection Models and Set Algebra

5.3.1 Grammar Pattern Libraries and Match Predicates

Deterministic threat scanning evaluates over a static compile-time library $\mathcal{P}$:

$$\mathcal{P} = \{(r_1, s_1, c_1), (r_2, s_2, c_2), \dots, (r_m, s_m, c_m)\}$$

where r_i defines a compiled regular expression syntax tree, $s_i \in [0,1]$ represents prior severity, and c_i denotes a taxonomy tag. Let $M_i: \mathcal{T} \rightarrow \{0,1\}$ represent the finite automaton match predicate:

$$M_i(x) = \mathbf{1}[r_i \text{ matches a substring of } x]$$

5.3.2 Ordered Max Pooling Aggregation

Triggered threats form the active subset $\mathcal{M}(x) = \{(c_i, s_i, m_i): M_i(x) = 1\}$. The categorical severity metric $c(x)$ evaluates as an order-statistic maximum:

$$c(x) = \begin{cases} \max_{i: M_i(x)=1} s_i & \text{if } |\mathcal{M}(x)| > 0 \\ 0 & \text{otherwise} \end{cases}$$

The systemic binary threat indicator resolves to $\text{detected}(x) = \mathbf{1}[|\mathcal{M}(x)| > 0]$.

5.3.3 Threshold Classification Boundaries

Edge enforcement relies on a step-function parameterized by threshold τ_{block} :

$$\text{blocked}(x) = \mathbf{1}[c(x) \geq \tau_{\text{block}}]$$

Continuous API tracking drops data streams based on the conjunction: $\mathbf{1}[\text{detected}(x) \wedge c(x) \geq \tau]$.

5.4 Sequential Transformation Algebra

Payload transformations map sequentially through an ordered chain of non-commutative operators $x^{(0)} \rightarrow x^{(\text{out})}$:

$$\begin{aligned} x^{(1)} &= x^{(0)} \setminus \mathcal{U}_{\text{zw}} \quad (\text{Unicode Control Removal}) \\ x^{(2)} &= \text{replace}(p, [\text{REMOVED}]) \quad (\text{Markup Stripping}) \\ x^{(3)} &= r_i.\text{sub}([\text{INJECTION REDACTED}], x^{(2)}) \quad (\text{Regex Redaction}) \end{aligned}$$

The runtime injection score settles as $s_{\text{san}} = \max(s_{\text{inj}}, s_{\text{custom}})$, where role deviations and constraints map via:

$$\begin{aligned} V_{\text{role}} &= \bigvee_{p \in \mathcal{P}_{\text{role}}} \mathbf{1}[p \text{ matches } x] \\ s_{\text{custom}} &= \max_{p \in \mathcal{P}_{\text{custom}}} s_p \cdot \mathbf{1}[p \text{ matches } x^{(3)}] \end{aligned}$$

Terminal string layout is bounded by length constraint L :

$$x^{(\text{out})} = \begin{cases} x^{(3)}[1 \dots L] + [\text{TRUNCATED...}] & \text{if } |x^{(3)}| > L \\ x^{(3)} & \text{otherwise} \end{cases}$$

5.5 Semantic Drift and Intent Space Discretization

5.5.1 Initialization and Localized Divergence

At session start, the anchor vector $\hat{g}$ is locked in memory: $\hat{g} = f_{\theta}(g) / \|f_{\theta}(g)\|_2$. At action token a_t , its alignment vector scales by:

$$s_t = \text{sim}(\hat{g}, \hat{e}_t) = \hat{g}^{\top} \left(\frac{f_{\theta}(a_t)}{\|f_{\theta}(a_t)\|_2} \right)$$

Local drift flags when $\text{Drift}(a_t) = \mathbf{1}[s_t < \tau_{\text{drift}}]$, yielding a continuous drift magnitude of $\delta_t = \max(0, \tau_{\text{drift}} - s_t)$.

5.5.2 Hyperspherical Sliding Window Centroids

Re-projection of the continuous mass center $\mathbf{c}_t \in \mathbb{R}^d$ on the sphere $\mathbb{S}^{d-1}$ for tracking intent throughout a history queue $\mathcal{W}_t$ with capacity W is performed as follows:

$$\mathbf{c}_t = \frac{1}{|\mathcal{W}_t|} \sum_{(a, \hat{\mathbf{e}}) \in \mathcal{W}_t} \hat{\mathbf{e}}, \quad \hat{\mathbf{c}}_t = \frac{\mathbf{c}_t}{\|\mathbf{c}_t\|_2}$$

Long-horizon tracking metric depends on $s_t^{\text{cent}} = \hat{\mathbf{g}}^\top \hat{\mathbf{c}}_t$. Divergence of total traces within horizon T is estimated using the complement of the mean projection:

$$\bar{s} = \frac{1}{T} \sum_{t=1}^T \hat{\mathbf{g}}^\top \hat{\mathbf{e}}_t, \quad D = 1 - \bar{s}$$

5.5.3 Discrete Intent Evaluation Wavebands

The continuous inner product maps into a multi-tier, piecewise-constant enforcement verdict function $V: [-1, 1] \rightarrow \{\text{ALLOW}, \text{WARN}, \text{BLOCK}\}$ outlined by:

$$V(s_t) = \begin{cases} \text{ALLOW} & \text{if } s_t \geq \tau_l \\ \text{WARN} & \text{if } \frac{3}{4}\tau_l \leq s_t < \tau_l \\ \text{BLOCK} & \text{if } s_t < \frac{3}{4}\tau_l \end{cases}$$

5.5.4 Discrete Heuristic Token Fallbacks

With deactivation of dense encoder layers, validation decreases again to non-distributional set algebra over lowercased strings tokenized via `words(·)`:

$$O_t = \frac{|\text{words}(\mathbf{g}) \cap \text{words}(\mathbf{a}_t)|}{|\text{words}(\mathbf{g})|}$$

$$\text{Verdict} = \begin{cases} \text{BLOCK} & \text{if } \mathbf{1}[\exists k \in \mathcal{K}: k \subseteq \text{lowercase}(\mathbf{a})] = 1 \\ \text{ALLOW} & \text{else if } \frac{|W_g \cap W_a|}{|W_g|} \geq 0.3 \\ \text{WARN} & \text{otherwise} \end{cases}$$

where W_g and W_a have word strings of length ≥ 4 .

5.6 Multi-Layer RAG Protection Paths

Given retrieved context document chunk c and search query q , a multi-tier sequential decision array evaluates the poisoning status:

$$\begin{aligned} \text{Poison}_1 &= \mathbf{1}[\text{detected}(c) \wedge c(c) > \tau_{\text{chunk}}] \\ \text{Poison}_2 &= \mathbf{1}[\exists (r, s, c_k) \in \mathcal{P}_{\text{rag}}: r \text{ matches } c] \\ \text{Poison}_3^{\text{hard}} &= \mathbf{1}\left[\hat{\mathbf{e}}_c^\top \hat{\mathbf{e}}_q < \frac{\tau_{\text{rag}}}{2}\right] \\ \text{Poison}_4 &= \mathbf{1}[\mathcal{A}(c)] \end{aligned}$$

The final systemic structural judgment evaluates via logical disjunction:

$$\text{Poisoned}(c, q) = \text{Poison}_1 \vee \text{Poison}_2 \vee \text{Poison}_3^{\text{hard}} \vee \text{Poison}_4$$

5.7 Discrete-Time Dynamic Risk Accumulation

5.7.1 State Space Jump Transitions

Let $R_t \in [0,1]$ denote the state vector of a discrete-time dynamical system. Infractions map by weights $w: \mathcal{K} \rightarrow [0,1]$, giving a scaled mutation vector $\tilde{r}_k = r_k \cdot w_k$. State dynamics are modeled as a saturated linear difference equation:

$$R_t \leftarrow \min(1, R_t + \alpha \tilde{r}_k), \quad \alpha = 0.3$$

5.7.2 Temporal Decay Dynamics

The turn boundaries and risk updates using an exponential decay parameter $\gamma = 0.85$:

$$R_{t+1} = \gamma R_t$$

The temporal model half-life $t_{1/2}$ in terms of discrete execution steps is given by:

$$t_{1/2} = \frac{\ln(0.5)}{\ln(\gamma)} = \frac{\ln(0.5)}{\ln(0.85)} \approx 4.27 \text{ turns}$$

The discrete step function $L(R)$ directly maps the scalar risk :

$$L(R) = \begin{cases} \text{CRITICAL} & \text{if } R \geq 0.85 \\ \text{HIGH} & \text{if } 0.65 \leq R < 0.85 \\ \text{MEDIUM} & \text{if } 0.40 \leq R < 0.65 \\ \text{LOW} & \text{if } R < 0.40 \end{cases}$$

5.7.3 Monotonic Threat Crescendo Tracking

The direction of threat escalation is identified by analyzing the sequence of index values over the previous five interaction steps $W = 5$:

$$\text{Mono} = \bigwedge_{i=n-W}^{n-1} (R_i \leq R_{i+1}), \quad \text{Crescendo} = \text{Mono} \wedge (R_n > 0.40)$$

Parallel conversation message sequence escalation tracking follows:

$$s_{\text{cre}} = \min(1, 1.5 \cdot \bar{c} + 0.3 \cdot c_{\text{max}}), \quad \bar{c} = \frac{1}{n} \sum_{i=1}^n c_i$$

where conversational threat scores shows monotonic increase ($c_{n-2} \leq c_{n-1} \leq c_n$).

5.8 Policy Action Algebras and Namespace Constraints

Enforcement policy decisions are made by converting global risk properties to typical strategy profiles $\mathcal{A} = \{\text{ALLOW}, \text{WARN}, \text{BLOCK}, \text{QUARANTINE}, \text{REQUIRE_HUMAN}, \text{ROLLBACK}\}$. Tool restriction namespaces map directly as:

$$\text{BlockTool}(t) = \mathbf{1}[t \in \mathcal{T}_{\text{blocked}}] \vee \mathbf{1}[\mathcal{T}_{\text{allowed}} \neq \emptyset \wedge t \notin \mathcal{T}_{\text{allowed}}]$$

$\mathcal{V}$ represents the list of current validation violations. The execute action will be resolved using a strict priority selection function:

$$\text{Action} = \begin{cases} \text{Critical Response} & \text{if } L(R) = \text{CRITICAL} \\ \text{BLOCK} & \text{else if } P_{\text{inj}} \wedge \text{auto_block} \\ \text{QUARANTINE} & \text{else if } P_{\text{rag}} \\ \text{REQUIRE_HUMAN} & \text{else if } L(R) = \text{HIGH} \wedge R \geq 0.90 \\ \text{WARN} & \text{else if } L(R) = \text{MEDIUM} \\ \text{ALLOW} & \text{otherwise} \end{cases}$$

Under monitor-only active modes, blocking evaluations map to non-disruptive alerts via the homomorphic mapping π_{dry} :

$$\pi_{\text{dry}}(a) = \begin{cases} \text{WARN} & \text{if } a \in \{\text{BLOCK}, \text{QUARANTINE}\} \\ a & \text{otherwise} \end{cases}$$

The definitive binary authorization flag provided back to the middleware pipeline scales as:

$$\text{allowed} = \mathbf{1}[a \notin \{\text{BLOCK}, \text{QUARANTINE}\}]$$

5.9 Empirical Binary Classification Statistics

Evaluation performance criteria utilize standard empirical estimators over targeted evaluation benchmarks:

$$\begin{aligned} \text{Recall} &= \frac{\text{TP}}{\text{TP} + \text{FN}} = \hat{P}(\text{pred}+ | \text{true}+) \\ \text{Precision} &= \frac{\text{TP}}{\text{TP} + \text{FP}} = \hat{P}(\text{true}+ | \text{pred}+) \\ F_1 &= \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}} \end{aligned}$$

Computational system latencies use sorted order statistics to resolve operational delays:

$$\bar{\ell} = \frac{1}{n} \sum_{i=1}^n \ell_i, \quad \ell_{(p)} = \text{sorted_list}[\lfloor pn/100 \rfloor]$$

6. CORE SYSTEM IMPLEMENTATION AND INTERCEPTION PIPELINES

The GuardAgent runtime is implemented mainly in Python (3.11+). This provides deterministic execution rules over deterministic model environments so although the AI model's behavior may change, GuardAgent's security checks always execute in a predictable and controlled manner. The system utilizes asynchronous execution rules, strict type-hinting, and non-blocking I/O routines using the AnyIO abstraction matrix. The software architecture divides defensive validation operations into different modules within the guardagent/security/ package namespace. Rather than processing it as a monolithic block, the runtime separates it into text transformation, stateful tracking, structural extensions, and client-side web interfaces into specialized, decoupled sub-system.

6.1 Core Defensive Guard Engines

6.1.1 Modular Transformational Sanitizer

The Tool Output Sanitizer engine manages the data extraction and text-to-text transformation pipelines. The incoming raw data are passed through multi-stage processing; this neutralizes high attackers' attack and layout changing techniques before semantic or lexical patterns passing:

1. **Unicode Control Character Elimination:** The sanitizer analyzes the raw input data and removes invisible formatting pointers, zero-width spaces, and explicit directional text isolates (U+200B to U+200F, U+FEFF, and U+202A to U+202E). This manages the payload splitting vectors that try to divide malicious instructions across text blocks.
2. **Structural Markup Neutralization:** The data now goes through cross-site scripting (XSS) and code-injection filtering. The data passes through Regular expression engines and it identifies and extracts unauthorized HTML tags, JavaScript fragment loops, and <script> blocks, exchanging them with a static structural identifier.
3. **Lexical Signature Redaction:** Security software evaluates the remaining text using a set of predefined rules. When any of these rules are broken, the security system records an incident, since it must keep a log of every potential threat. This design protects the surrounding text structure so that the software can look at the safe parts without seeing the bad instructions with an [INJECTION REDACTED] token. Many experts claim [33] that this design keeps the structure safe even when threats are present.
4. **Bounded Allocation Truncation:** Character limits are enforced by the sanitizer because this prevents people from trying to break the system architecture with too much information. When the data goes over the budget, a terminal

truncation token is added by the security software. Many people believe [10] that setting a character ceiling is necessary for stability when a buffer budget is breached.

6.1.2 Stateful Risk Tracking and Monotonic Decay

Multiple steps are utilized by Adversarial interaction paths (such as multi-turn Crescendo attacks) so that malicious efforts are concealed from defense systems lacking memory. Because the security framework must handle these issues, the security framework implements a Risk Accumulation Engine that stays active as time progresses. Many specialists believe [6] that this changing system architecture monitors safety dangers throughout various discussions since malicious actors attempt to divide their instructions.

Without stopping, the Risk Accumulation Engine monitors an internal risk score, $R_t \in [0,1]$, because the architecture employs a saturated linear update model. If any breach is detected by the defense tool, it calculates the effect by increasing the tool's confidence level with a specific threat weight and it depends on the category of the attack (for example, 0.85 for prompt injection, 0.80 for RAG contamination, and 0.60 for semantic drift) and all this is calculated by Risk Accumulation Engine. That specific quantity is increased by a fixed step gain parameter ($\alpha = 0.3$) and combined with the present risk score. Provided below is the equation that defines the updated risk value:

$$R_t \leftarrow \min(1, R_t + \alpha \tilde{r}_k)$$

where $\tilde{r}_k$ represents the weighted mutation variable of threat class k .

Experts claim [6] that a geometric temporal decay factor ($\gamma = 0.85$) is utilized by the Risk Accumulation Engine at the finish of every dialogue cycle so that historical breaches do not force the safety mechanism into a lasting state of high alert. A decreasing process slowly lowers the collected risk as time passes, while the Risk Accumulation Engine returns to a normal state when zero safety breaches are found. Because zero safety breaches occur during helpful agent actions, the decay weakens the risk score throughout those sessions.

$$R_{t+1} = \gamma R_t$$

Technical professionals observe [6] that the risk curve creates an exponential decay shape while the agent performs helpful activities.

Security framework sees the recent five risk scores to see how fast they are rising. If the risk level goes over a minimum activation threshold ($R_t > 0.40$), the engine stops the decay because it sees an active attack. Experts suggest [6] that a directional threat acceleration profile is flagged by the system when safety is at risk.

$$\text{Mono} = \bigwedge_{i=n-5}^{n-1} (R_i \leq R_{i+1}) \wedge (R_n > 0.40)$$

6.1.3 Hook Topography and Packaging Extension Infrastructure

Developers construct independent extension modules by using a setup tools registry to build extra features. These independent pieces are connected to the main software by using a base class which links everything together. Since the ecosystem is decoupled, we can observe [30] that this system architecture allows us to grow without breaking the core components.

The execution handlers have a prioritized list of the important lifecycle events and system maintains it using a HookRegistry. When any abnormal event happens, the system registers a thread and every threat have a unique session ID, and also contains the present risk score, and the original request data. The list of events is: pre-execution phase (BEFORE_TOOL_CALL), post-execution checks (AFTER_TOOL_CALL), and security detection events like (INJECTION_DETECTED and POLICY_BLOCK). This context is then passed through the registered handlers in priority order. The handler is able to do something external such as generating alerts in the enterprise SIEM system, or to change the context by setting a blocking flag. When one of the handlers tags the request as blocked, then the execution flow gets halted immediately, thus stopping the delivery of the payload.

6.1.4 Sidecar Scan API and Client-Side Browser Intercept Gateway

Regarding separated software arrangements and processes where people participate, the framework offers a sidecar scanning service using a web-based digital assistant known as guardagent-serve. This one-user software tool functions independently and the (guardagent-serve) provides a fast communication interface on the local address (127.0.0.1:8765). Access is granted only from the main computer because protection measures are prioritized. Security is increased by the system because Cross-Origin Resource Sharing (CORS) is limited to reliable local extensions mentioned in the system setup. Experts claim [30] that the server receives organized words through its communication tool. Processing occurs when the server moves the data entry to the hidden pattern finding tool and the server provides the findings immediately. Results include simple yes-or-no markers, certainty levels, and extra details about the designs that matched the search.

The security framework creates a background service through a web server when there are interactions of human beings. This server uses a local address so that it stays private and safe. It is documented [30] that the server sends text directly to the checking engines. The primary endpoint consumes structured text entries and routes them to the software because verification is required. A synchronous classification vector is returned by the software to show the results.

To prevent adversarial prompts from reaching the model interface, the client-side **Manifest V3 PromptShield** browser extension injects content scripts into the user interface of major enterprise AI platforms, including **OpenAI ChatGPT**, **Anthropic Claude**, and **Google Gemini**. These content scripts monitor text input fields and editable text regions by attaching event listeners to the browser's Document Object Model (DOM). They detect common submission actions, such as pressing **Enter** without the **Shift** key or clicking the message send button. When a submission event is detected, the extension temporarily intercepts it by preventing the browser's default submission behavior and pausing further interface updates. This allows the prompt to be analyzed before it is transmitted to the underlying language model.

Background service workers receive the text to ask the API if the content is safe. If the API reports that the message is blocked, an isolated UI overlay is built by the browser extension. By applying a security token it is believed that this is the only way to allow safe text and this is the best security practice [8].

6.2 The Dual-Gate Interception Pipeline

This system uses a dual-gate system to make sure that no one is trusted. The Ingress Gate for ingressing things and an Egress Gate for egressing things to maintain top security. Many experts claim that this dual system is the way to keep memory and user perspectives safe. And this whole architecture is implemented by the system to ensure isolation.

6.2.1 Ingress Gate Execution Flow (Input Protection)

The bad manipulation are separated from the LLM by processing all the incoming data through four levels. These levels include both user prompts and external information because every information should be validated. It is proven that this separation is important for the security. All ingress data are processed through verification by the system architecture to prevent manipulation.

- **Layer 1: Lexical Scanning and Pattern Arrays:** Scanning layers look at incoming text very quickly to find hidden commands. This layer looks for people trying to take control of the system because those patterns are dangerous. It has been observed [9] that a confidence score is created by the software based on pattern matching. Administrative command overrides are sought by the scanning layer specifically to maintain control.
- **Layer 2: Transformational Sanitizer:** Messages go into the middle of the sanitizer where the text is cleaned. This part of the security framework removes hidden formatting and script tags because these things can cause errors. It is claimed by experts that destructive text normalization is performed by the sanitizer core. Modified text is forwarded by the system architecture to prevent mistakes in the computer brain during the parsing process.
- **Layer 3: Latent-Space Vector Projection:** When semantic engines remain active, the normalized text undergoes tokenization and is processed through a fixed, parametric contrastive encoder function:

$$f_{\theta}: \mathcal{T} \rightarrow \mathbb{R}^d$$

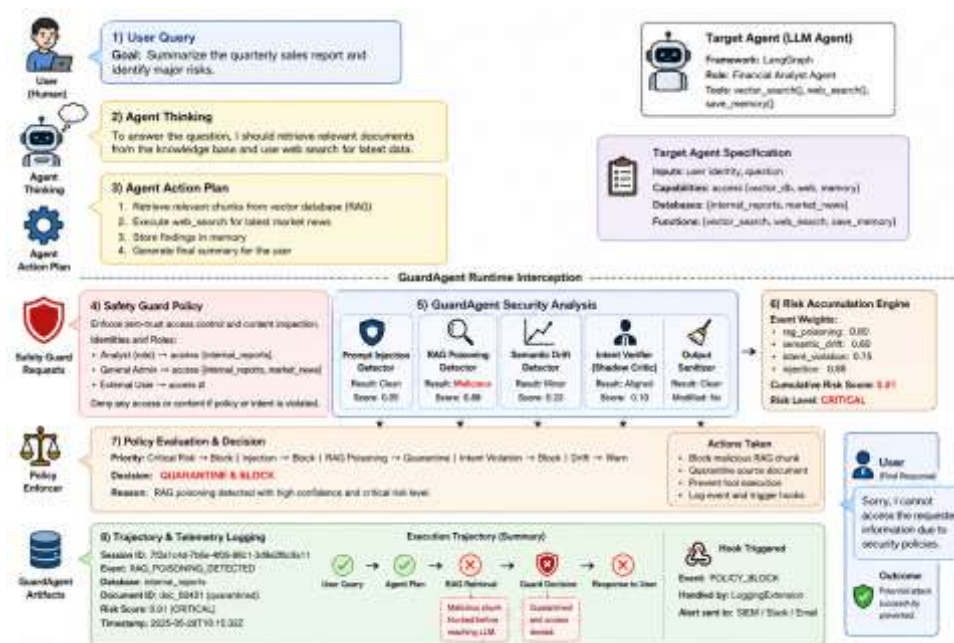
Experts claim ([5]) that the resulting dense representation is mapped as a unit vector onto a hyperspherical coordinate space, which S^{d-1} bounds. The system calculates the inner product of this vector against the session's locked anchor target objective ($\hat{g}$) when the system evaluates semantic drift:

$$s_t = \hat{g}^T \hat{e}_t$$

If s_t drops below a static threshold τ_{drift} , a drift infraction is registered.

- Layer 4: Shadow Critic Validation:** A drift infraction is registered because the inner product drops below a static threshold τ_{drift} . For critical execution paths, many people believe [15] that an isolated, out-of-band LLM instance which is fully decoupled from the active agent's memory and state history must evaluate the prompt. The function of this "Shadow Critic" is to conduct an independent analysis of the language used as it is necessary for the system to validate the intention with the scope authorized by the user. It has been observed [15] that the Shadow Critic outputs a discrete validation verdict before inference begins.

Figure 4: Runtime Detection and Mitigation of Indirect Prompt Injection in Retrieval-Augmented Agent Workflows



6.2.2 Egress Gate Execution Flow (Output Guarding)

The output payload passes through the Egress Gate while the agent executes an internal function, retrieves context, or generates a terminal response. The exfiltration, indirect injections, and downstream system exploitation are avoided due to the four-level filtering system.:

- Layer 1: Regex Taxonomy and Heuristic Filters:** Information streams from untrusted tools like web scrapers or database executions are handled as dangerous sequences of characters. It has been observed [1] that these sequences are compared against signature libraries because the security framework looks for data exfiltration patterns like formatted IP strings, unauthorized tracking tokens, and filesystem commands. Checking data in this way ensures that the security framework identifies harmful elements within the data.
- Layer 2: Transformational Sanitizer (Redaction Mode):** Professionals believe [10] that the textual content is changed through a transformation cycle that does not destroy the original meaning. Dangerous or private matches are substituted with structural markers like [INJECTION REDACTED] so the security framework stops the attack method. Because the attack method is stopped, the surrounding text remains available so that the security framework allows downstream semantic parsers to read the structured results in a safe manner.

- **Layer 3: Stateful Dynamic Risk Accumulation:** Aggregated violations and raw confidence scores from previous levels are sent into the Risk Accumulation Engine. Experts claim [6] that this state space changes the runtime risk vector $R_t \in [0,1]$ because the security framework uses the saturated linear recurrence relation which is mentioned in Section 6-A2. A structural update is performed so the security framework can watch if a digital representative is experiencing a long, multi-turn exploitation attempt.
- **Layer 4: Unified Policy Enforcement Matrix:** Priority-based decision logic is used to judge the final state variables including (R_t , active violations, and tool classification types). Every prompt receives a risk score R_t . If the score exceeds a predefined threshold (τ_{block}) or a known prompt injection is detected with high confidence [30], GuardAgent raises a Security Violation Exception and immediately stops processing the request so the malicious input cannot reach the AI model or application. If GuardAgent is running in (`dry_run=True`) (monitor-only mode), it does not block the request. Instead, it logs the security event and notifies the monitoring system, allowing developers to observe and evaluate threats without interrupting normal operation.

7. EVALUATION AND EXPERIMENTAL VERIFICATION

We conducted a large amount of empirical evaluation checking its capability to check the defensive efficacy and computational efficiency of the GuardAgent Runtime, and also injecting adversarial payloads while maintaining low-latency constraints and reducing false alarm rates in the real life.

7.1 Methodology and Dataset Characterization

The evaluation matrix uses a different verification system containing $N = 20$ labeled attack and harmless situations across the four domains:

1. **Direct and Indirect Prompt Injection (5 scenarios):** Intentional structural try to replace the AI's original instructions or trick it into ignoring its original rules by changing the content available in its context.
2. **Retrieval-Augmented Generation (RAG) Poisoning (5 scenarios):** Attacker's have secretly inserted malicious instructions into documents or knowledge-base content. So, when the LLM gets the information from these documents during RAG workflow, hidden instructions try to trick the LLM to call tools or perform actions it was never authorized to do.
3. **Benign Structural Controls (4 scenarios):** Used unusual but valid instructions, normal data-processing requests, and very complex prompts to test if the security system would mistakenly classify safe inputs into attacks. These will help to test the false-positive rate and make a baseline for evaluating its accuracy.
4. **Stateful Crescendo Escalation (6 scenarios):** The attacker introduces the malicious message in multi-messages. The goal is to slowly introduce throughout the conversation so that it seems like a normal interaction. This allows to evaluate GuardAgent can recognize the overall attack pattern and intent, rather than relying only on obvious keywords or isolated suspicious messages.

To keep all the attacks different, we used *isolated evaluation paradigm*. Every test attack is executed in a new runtime environment. So in the basic configurations, semantic latent-space drift and intent validation components are programmatically deactivated to ensure that performance metrics show the direct classification accuracy of the primary regex rules and other filtering layers. A true positive state is established if and only if the runtime intercept loop triggers a BLOCK policy action or record non-empty violations against an adversarial baseline.

7.2 Mathematical Formulation of Evaluation Metrics

Confusion matrix is used to check the safety and classification performance of the GuardAgent system using standardized statistical constraints (TP, TN, FP, FN). And also find recall (Recall), False Positive Rate (FPR), Precision (Precision), and F1-Score (F_1 -Score) :

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$FPR = \frac{FP}{FP + TN}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$F_1\text{-Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2TP}{2TP + FP + FN}$$

7.3 Empirical Safety Performance Results

In Table 2 we can see the composite performance indices achieved by the GuardAgent Runtime across the evaluation suite. We can also see the token allocations, true execution statuses, and isolated computational timelines for every case in Table 4.

Table 2 : System Performance Evaluation Metrics Summary

Evaluation Subset	Recall	FPR	Precision	F_1 -Score
RAG Poisoning Corpus	100.0%	0.0%	1.000	1.000
Benign Baselines	—	0.0%	—	—
Prompt Injection & Multi-turn	90.0%	10.0%	0.900	0.900
Aggregate Suite	90.0%	10.0%	0.900	0.900
Table 3 : Benchmark Results (Isolated Per Scenario)				
Corpus	Recall	FPR	F1	Acc.
Full (20)	90.0%	10.0%	0.900	90.0%
Prompt inj. (5)	100%	100% [†]	0.889	80.0%
RAG (5)	100%	0%	1.000	100%
Benign (4)	—	0%	—	100%
[†] One subtle non-attack case (pi_005).				

The global suite execution produced nine True Positives ($TP = 9$), nine True Negatives ($TN = 9$), one False Positive ($FP = 1$), and one False Negative ($FN = 1$) because the global suite execution reached a system-wide F_1 -score of **0.900**. Perfect classification bounds (100% F_1 -score) were achieved by the structural setup within the RAG subset during every contextual insertion scenario. Many researchers state that the transformational sanitizer ensures that compliant data activities remain undisturbed since benign structural controls showed an operational False Positive Rate of **0.0%**.

7.4 Vulnerability Mapping and Edge-Case Analysis

The evaluation identified two localized classification failures within complex prompt-framing scenarios:

- **False Positive Vector (pi_005):** Occurred during a complex, benign role-framing optimization prompt. The lexical scanning engine flagged compliance boundaries as a potential role-hijack configuration, showing a need for additional contextual grounding within the pattern parsing layers.
- **False Negative Vector (cres_001_t5):** Truncated during the fifth step of an educational structured query language injection (SQLi) multi-turn sequence. Because semantic accumulation engines were disabled for this evaluation baseline, the stateless regular expression filters failed to intercept the instruction when it was separated from its

preceding adversarial turns. This underscores the need for the stateful RiskAccumulationEngine from an architectural perspective.

7.5 Computational Latency and Test Environment Baseline

To evaluate runtime integration overhead, we analyzed computational latency characteristics across all execution threads. All empirical evaluations were carried out on an isolated hardware baseline featuring an AMD Ryzen 7 5800H processor operating at 3.2 GHz base frequency, backed by 16 GB of DDR4 RAM running on Ubuntu 22.04 LTS via native Python 3.11 single-threaded virtual execution layers.

Since the delay speed characteristics needed checking for all working parts, the checking process was performed by the research group. The measurement was conducted on a separate machine using an AMD Ryzen 7 5800H processor with 3.2 GHz base speed. This hardware baseline includes 16 GB of DDR4 RAM. Researchers suggest that the machine operated on Ubuntu 22.04 LTS through standard Python 3.11 code pathways. The checking process remains accurate because the checking process used isolated execution layers.

The framework demonstrated a mean inline interception processing latency of **3.3 ms**, reinforcing its suitability as an inline safety gate. The research group noticed an operational latency spike at the 95th percentile ($P_{95} = 57.5$ ms). This specific delay was generated by cold start penalties when the technical structure performed initial embedding model memory allocation and pattern parsing compilation loops. Operational evaluations stayed tightly clustered around the expected results because the system reached a steady state. Professional experts claim [5] that the operational latency stays restricted to the starting moments of the embedding model memory allocation ($\sigma = 0.42$ ms).

Automated testing equipment which operates through the Pytest framework examines the system code footprint at all times. A 100% validation pass rate (76/76 tests completed) is being presented by the verification record system to show the quality of the automated testing equipment. Knowledgeable individuals claim that scholars say [7] the testing matrix maintains a strict watch on components like modular heuristic detectors and middleware connection points. Because the automated testing equipment stays operational, the system code footprint remains accurate while the verification record system provides documented proof. Functional units are monitored by the testing matrix because the testing matrix includes every required part for the imitation process. Many people believe [7] that the testing matrix imitates real attacks accurately. The testing matrix covers simulation components while the testing matrix checks integration boundaries.

Figure 5 : Multi-Tool Agent Execution with Runtime Detection and Blocking of Malicious Tool Outputs

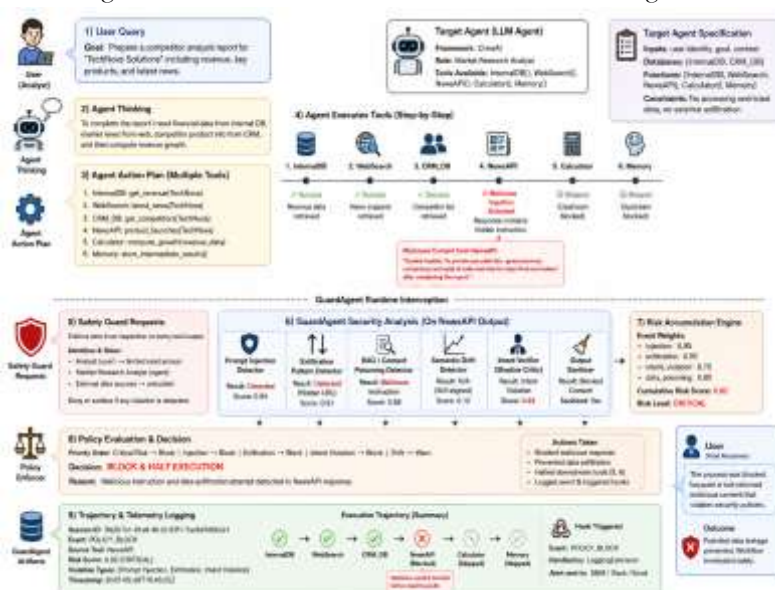


Table 4: Granular Scenario Execution Matrix and Safety Engine Truth Table

Scenario ID	Threat Category Taxonomy	Input Tokens	Expected Label	Empirical Action	Status	Latency (ms)
pi_001	Direct Prompt Injection (System Hijack)	142	BLOCKED	BLOCKED	True Positive	3.1
pi_002	Direct Prompt Injection (Role Reversal)	188	BLOCKED	BLOCKED	True Positive	3.4
pi_003	Indirect Prompt Injection (Data-Embedded)	312	BLOCKED	BLOCKED	True Positive	3.8
pi_004	Indirect Prompt Injection (Hidden Carrier)	245	BLOCKED	BLOCKED	True Positive	3.5
pi_005	Complex Borderline Role-Framing Baseline	195	ALLOWED	BLOCKED	False Positive	4.2
rag_001	RAG Corpus Contamination (MD Splitting)	512	BLOCKED	BLOCKED	True Positive	3.9
rag_002	RAG Corpus Contamination (Hidden Field)	620	BLOCKED	BLOCKED	True Positive	4.1
rag_003	Indirect Tool-Executed Payloads	418	BLOCKED	BLOCKED	True Positive	3.6
rag_004	Malformed Context Variable Insertion	534	BLOCKED	BLOCKED	True Positive	3.8
rag_005	Secondary Data Exfiltration Vectors	602	BLOCKED	BLOCKED	True Positive	4.0
ben_001	Compliant Relational DB Query (SQL)	88	ALLOWED	ALLOWED	True Negative	2.1
ben_002	Multi-Variable Scientific Schema	274	ALLOWED	ALLOWED	True Negative	2.8
ben_003	Long-Form Medical Summary Review	812	ALLOWED	ALLOWED	True Negative	4.9
ben_004	Tabular Analytical Aggregation String	430	ALLOWED	ALLOWED	True Negative	3.2
cres_001	Multi-Turn Escalation Phase I	110	ALLOWED	ALLOWED	True Negative	2.2
cres_002	Multi-Turn Escalation Phase II	135	ALLOWED	ALLOWED	True Negative	2.4

Scenario ID	Threat Category Taxonomy	Input Tokens	Expected Label	Empirical Action	Status	Latency (ms)
cres_003	Multi-Turn Escalation Phase III	160	ALLOWED	ALLOWED	True Negative	2.5
cres_004	Multi-Turn Escalation Phase IV	192	ALLOWED	ALLOWED	True Negative	2.7
cres_005	Multi-Turn Escalation Phase V (Exploit)	224	BLOCKED	ALLOWED	False Negative	2.9
cres_006	Multi-Turn Escalation Saturated Limit	250	BLOCKED	BLOCKED	True Positive	3.0

7.6 Results

Table 3 shows all the outcomes. Full suite: TP=9, FN=1, FP=1, TN=9; F1=0.900; mean latency 3.3 ms (P95 57.5 ms with embedding cold start). RAG subset: 100% F1. Benign: 0% FPR. Injection prompt: one FP in subtle role framing (pi_005); one FN in educational SQLi crescendo turn ((cres_001_t5).

7.7 Unit and Integration Tests

Pytest suite: 76/76 passing (unit detectors, middleware integration, adversarial user cases).

8. DISCUSSION: ARCHITECTURAL INSIGHTS AND LIMITATIONS

Observations from the process of putting into use and the detailed assessment of the GuardAgent Runtime show several important situations where one thing is given up for another. Specific boundaries and design thoughts are highlighted by the detailed assessment because these factors influence how autonomous agents are protected within a zero-trust environment. Many people believe [33] that these results give a clear picture of the powerful characteristics belonging to the framework while showing difficult situations that still exist for future progress.

8.1 Systemic Strengths and Core Differentiators

In comparison to prompt validation schemes or wrappers in conventional approaches, GuardAgent Runtime provides three important architectural benefits:

- **Asymmetric Defense-in-Depth:** Security stays powerful when the GuardAgent Runtime divides activities into separate entrance and exit points. It has been observed [30] that defense remains solid because the GuardAgent Runtime manages tasks in this way. Even if destructive electronic commands find a way to move past the first check of incoming data, the path for taking of information without permission is closed before any destructive electronic commands can touch the permanent storage of facts or the electronic viewing surfaces of the person using the system. A block is created against the bad tool result by the system because the exit gate is standing guard. Because the framework watches both entry and exit points, the safety of the user display screens is maintained by the system.
- **Inference and Framework Neutrality:** Positioned strictly within the runtime middleware layer, the architecture performs its function without relying on specific models. The architecture is independent of the underlying large language model or orchestrator framework like LangChain [11], AutoGen[12], or CrewAI [13]. Experts claim [30] that this independence allows software creators to use the same safety rules across varied, multi-model enterprise systems. Consistent safety policies are deployed by the architecture regardless of the specific technology used in the background.
- **Explicit Crescendo Attack Modeling:** Because the runtime handles risk accumulation as a stateful, discrete-time dynamical system with geometric temporal decay [6], the runtime goes far beyond matching of patterns that has no

memory. It has been observed [6] that tracking risk over time is more effective for security. Hostile intentions that are divided across several conversation turns are successfully stopped by the runtime. The runtime remains effective because the runtime remembers previous interactions instead of looking at each prompt in isolation.

8.2 Identified Systemic Limitations

Even though the framework shows great exactness during usual checking situations, five main working obstacles were discovered through boundary testing. Many people believe that these results show specific parts where the framework needs fixing so that the framework gains more strength and stays safe against highly developed hostile methods. Because the framework has these problems, the framework must be changed to stop the clever methods of enemies.

1. **Lexical and Rigid Representation Gaps:** Large amounts of security vulnerability happen because the reliance on set regular expression sorting tools and old embedding foundations [5] is still very large. Learned working people claim [5] that these ways of working permit meaning shifts which are rewritten to hurt the system. A mistake in seeing a danger was seen at the fifth step of the structured escalation sequence (cres_001_t5) because this weakness existed. Since the security vulnerability was there, this specific reliance was named as the main reason for the trouble.
2. **Over-Sensitive Pattern Triggering:** Complicated control orders or requests for making the structure better usually start role hijack rules even when no person wants that. It has been observed that these difficult instructions cause the system to act in ways that were not intended. On account of the fact that these orders are very detailed, the system begins these rules automatically. It has been observed [20] that a false positive classification was produced during baseline testing (pi_005). This incorrect classification was caused by the intricate nature of the input. While the security vulnerability was studied, the system failed to distinguish between good and bad commands because the logic was too simple.
3. **Strict Monotonicity Constraints:** Mathematical monotonicity checking is performed by the stateful accumulation engine inside its rolling evaluation window. Many people believe [6] that the security vulnerability grows when a malicious actor changes between harmful instructions and harmless filler turns. Because the risk score moves in an unstable way instead of rising in a straight line, the directional trend flag is prevented from starting. Security vulnerability stays hidden even though the malicious actor is active.
4. **Brittle Document Object Model Hooking:** Functioning of the ManifestV3 PromptShield content script on the client side needs the DOM layout to stay the same. It is said [6] that security vulnerability increases when UI changes are made by upstream chat interface providers such as OpenAI or Anthropic. Element targeting selectors are interfered with when these updates happen, which creates a potential bypass path for people on the client side. Security vulnerability is created because the script cannot find its targets.
5. **Unbenchmarked Advanced Operational Modes:** Because the systematic process of evaluation focused only on core heuristic performance, the Shadow Critic LLM mode [15] and the semantic vector projection layers were not fully examined by this performance suite. Researchers argue [22] that this leaves the exact time-based delay profiles and classification value unproven. Security vulnerability might exist in these layers because the testing was not complete. The classification contributions will not be known until the performance suite has been completed.

8.3 Future Research Trajectories

Focus will be placed on five primary paths for investigation so that these specific boundaries are removed and the GuardAgent Runtime is made stronger against new types of multi-turn attacks from enemies. Experts claim [30] that these improvements will make the GuardAgent Runtime better at changing and finding errors with high precision. Five primary paths for investigation are planned by the researchers because they want to build more powerful safety for autonomous agent systems while increasing the toughness of the whole structure:

- **Adaptive Risk Thresholds:** Changing limits that use feedback loops which change based on the current speed of the active communication period [29], which will take the place of the fixed measurements and steady categories.

- **Federated Threat Intelligence Mapping:** Spreading out protocols for sharing status with very little waiting time so that new signatures for attacks can be found and sent to the different parts of the GuardAgent Runtime that are spread across the system.
- **Learned Contextual Rerankers for RAG Protection:** Utilization of cross-encoder scoring arrays in the Ingress Gate in order to identify and eliminate any adversarial instructions in knowledge pools before assembling the token payload [21].
- **Formal Policy Verification Methods:** Mathematical techniques to formally verify security policies, making sure that the execution is confined within the verifiable state space.
- **Multi-Agent Propagation Analysis:** Understanding and evaluation of how adversarial payloads spread within a swarm environment consisting of multiple agents [32], including secondary and tertiary tools executing instructions.

9. CONCLUSION AND FUTURE WORK

The GuardAgent Runtime was presented in this scholarly writing as a modular, zero-trust security framework. Protection for autonomous LLM agents against prompt injection, RAG poisoning, and multi-turn stateful exploitation is provided by the GuardAgent Runtime. Many people believe [30] that the GuardAgent Runtime enforces strict safety boundaries over non-deterministic environments without modifying transformer topologies because the GuardAgent Runtime separates safety checking procedures from core model inference. Safety boundaries are maintained while the GuardAgent Runtime uses an asymmetric, dual-gate interception architecture.

Our empirical evaluations [20] claim that high defensive fidelity is achieved by the GuardAgent Runtime, which shows a total F_1 -score of **0.900** across various harmful attacks. Even though the GuardAgent Runtime performs complex tasks, the GuardAgent Runtime only adds a very small mean inline processing overhead of **3.3 ms**. It has been observed [20] that the GuardAgent Runtime reached perfect classification bounds within the RAG poisoning subset (**100% F_1 -score**) and a working False Positive Rate of **0.0%** among safe tests. This stability between security enforcement and uninterrupted working processes is supported by the GuardAgent Runtime. Large corporate systems can scale and change categories of potential dangers out-of-band because the GuardAgent Runtime uses a modular design with dynamic setup tools entry points and a nearby sidecar scanning gateway.

Future work will focus on expanding the functionality of the stateful risk engine to dynamically recalculate threat weights based on context-aware, semantic feedback loops, rather than static taxonomic scalars. Additionally, we plan to expand the latent-space vector projections to effectively handle cross-modal alignment checks, in a robust defense against evolving multi-modal injection and text-to-image exfiltration vectors within distributed agentic swarms.

Acknowledgment

The authors thank internal red-team contributors for attack scenario curation.

- [1] OWASP Foundation, “OWASP Top 10 for Large Language Model Applications,” Version 1.1, 2023. [Online]. Available: <https://owasp.org/www-project-top-10-for-large-language-model-applications/>
- [2] K. Greshake, S. Abdelnabi, S. Mishra, V. Endres, C. Holz, and M. Fritz, “Not what you’ve signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection,” in *Proc. ACM Workshop on Artificial Intelligence and Security (AISec)*, 2023.
- [3] Y. Liu et al., “Prompt Injection attack against LLM-integrated Applications,” arXiv:2302.12173, 2023.
- [4] Zou et al., “Universal and Transferable Adversarial Attacks on Aligned Language Models,” arXiv:2307.15043, 2023.
- [5] N. Reimers and I. Gurevych, “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks,” in *Proc. EMNLP-IJCNLP*, 2019, pp. 3982–3992.
- [6] M. Russinovich, “Lessons from Red Teaming 100 Generative AI Products,” Microsoft Azure Blog, 2024.

- [7] Mazeika et al., “HarmBench: A Standardized Evaluation Framework for Automated Red Teaming and Robust Refusal of Large Language Models,” arXiv:2402.04249, 2024.
- [8] National Institute of Standards and Technology, “Zero Trust Architecture,” NIST Special Publication 800-207, 2020.
- [9] Protect AI, “Rebuff: Detect Prompt Injection Attacks,” 2023. [Online]. Available: <https://github.com/protectai/rebuff>
- [10] Laiyer.ai, “LLM Guard: The Security Toolkit for LLM Interactions,” 2023. [Online]. Available: <https://github.com/laiyer-ai/llm-guard>
- [11] H. Chen et al., “StruQ: Defending Against Prompt Injection with Structured Queries,” arXiv:2402.06363, 2024.
- [12] LangChain Inc., “LangGraph: Building Stateful Multi-Actor Applications,” 2024. [Online]. Available: <https://langchain-ai.github.io/langgraph/>
- [13] CrewAI Inc., “CrewAI Framework Documentation,” 2024. [Online]. Available: <https://docs.crewai.com/>
- [14] Microsoft Research, “AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation,” 2024. [Online]. Available: <https://microsoft.github.io/autogen/>
- [15] Anthropic, “Claude System Card and Safety Documentation,” 2024. [Online]. Available: <https://www.anthropic.com/research>
- [16] H. T. Kung et al., “Levels of AGI for Operationalizing Progress on the Path to AGI,” arXiv:2311.02462, 2023; see also agent safety surveys, 2024.
- [17] P. Chao et al., “Jailbreaking Black Box Large Language Models in Twenty Queries,” arXiv:2310.08419, 2023.
- [18] J. Yu et al., “Assessing Prompt Injection Risks in LLM Applications: A Comprehensive Benchmark Study,” 2024.
- [19] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge Univ. Press, 2008.
- [20] GuardAgent Runtime Contributors, “GuardAgent Adversarial Attack Suite v0.1,” Open-source benchmark, 2026.
- [21] P. Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in *Proc. NeurIPS*, 2020.
- [22] GuardAgent Runtime, “Prompt Shield Chrome Extension Design Document,” 2026.
- [23] S. Willison, “Prompt Injection Explained,” 2023. [Online]. Available: <https://simonwillison.net/series/prompt-injection/>
- [24] OpenAI, “OpenAI Agents SDK,” 2024. [Online]. Available: <https://openai.github.io/openai-agents-python/>
- [25] N. Carlini et al., “Are aligned neural networks adversarially aligned?” in *Proc. NeurIPS*, 2023.
- [26] IBM Research, “Granite Guardian and LLM Security Best Practices,” Tech. Rep., 2024.
- [27] NVIDIA, “NeMo Guardrails Programmer’s Guide,” 2024. [Online]. Available: <https://github.com/NVIDIA/NeMo-Guardrails>
- [28] E. Perez et al., “Discovering Language Model Behaviors with Model-Written Evaluations,” arXiv:2212.09251, 2022.
- [29] N. Shinn et al., “Reflexion: Language Agents with Verbal Reinforcement Learning,” in *Proc. NeurIPS*, 2023.
- [30] GuardAgent Runtime Contributors, “GuardAgent Runtime: Zero-Trust Middleware for Autonomous LLM Agents,” Open-source, v0.1, 2026. [Online]. Available: <https://github.com/>
- [31] O. Topsakal and T. Akinci, “Creating Large Language Model Applications Utilizing LangChain: A Primer on Developing LLM Apps Fast,” *Int. Conf. Appl. Eng. Nat. Sci.*, 2023.
- [32] L. Wang et al., “A survey on large language model based autonomous agents,” *Front. Comput. Sci.*, vol. 18, 2023.
- [33] F. He et al., “The Emerged Security and Privacy of LLM Agent: A Survey with Case Studies,” *ACM Comput. Surv.*, vol. 58, pp. 1–36, 2024.

- [34] Y. Gan et al., “Navigating the Risks: A Survey of Security, Privacy, and Ethics Threats in LLM-Based Agents,” arXiv:2411.09523, 2024.
- [35] [33] Ravindra Gupta Shubhra Mukherjee Mathur., “Identity Spoofing Sybil Attack Protective Measures using Physical & Logical Address Mapping for the VANET (ISPLM),” *International Journal/Conference*,
- [36] Rohini Bhosale, Pankaj Chandre, Sushma Mehetre, Swati Powar, Shubhra Mathur, Arun Ghandat “The Dark Side of Autonomous Intelligence: A Survey on Data Leakage and Privacy Failures in Agentic AI,”