SOK: A TAXONOMY OF ATTACK VECTORS AND DEFENSE STRATEGIES FOR AGENTIC SUPPLY CHAIN RUNTIME

Shiqi Yang[1,†], **Wenting Yang**[1,†], **Xiaochong Jiang**[1,†], **Yichen Liu**[1], **Cheng Ji**[1]
[1]Independent Researcher
sy3506@nyu.edu, wey023@ucsd.edu, jiang.xiaoc@northeastern.edu, yil160@ucsd.edu, chengji5@illinois.edu *

ABSTRACT

Agentic systems based on large language models (LLMs) operate not merely as text generators but as autonomous entities that dynamically retrieve information and invoke tools. This execution model shifts the attack surface from traditional build-time artifacts to inference-time dependencies, exposing agents to manipulation through untrusted data and probabilistic capability resolution. While prior work has examined model-level vulnerabilities, security risks arising from the complex, cyclic runtime behavior of agents remain fragmented.

This paper systematizes existing research into a unified runtime framework. We categorize threats into data supply chain attacks (distinguishing between transient context injection and persistent memory poisoning) and tool supply chain attacks (spanning discovery, implementation, and invocation phases). Crucially, we identify the emergence of the Viral Agent Loop, where agents effectively become vectors for self-propagating generative worms that require no code vulnerabilities to spread. We argue for a transition to a Zero-Trust Runtime Architecture, where context is treated as untrusted control flow, and tool execution is bounded by cryptographic provenance rather than semantic likelihood.

Keywords: Agentic AI, LLM Agents, Supply Chain Security, Indirect Prompt Injection, Runtime Security

1 INTRODUCTION

Large Language Models are increasingly deployed as autonomous agentic systems, moving beyond passive text generation. Modern agents retrieve information from external sources, maintain memory across interactions, and invoke tools capable of directly modifying digital or physical states Wang et al. (2023); Xi et al. (2025). As a result, agent behavior is influenced not only by model parameters and developer prompts, but also by dynamically acquired information and capabilities during execution.

This execution model establishes a fundamentally different security posture, referred to as Stochastic Dependency Resolution. Unlike traditional software systems, where dependencies such as `import numpy` resolve to a specific binary hash prior to deployment, agentic systems assemble their execution context at runtime based on semantic probability. Consequently, external documents, retrieved knowledge, APIs, and tools become implicit dependencies. Inference-time context therefore operates as an active component of the system's attack surface, rather than serving as a passive input.

This shift challenges established cybersecurity assumptions. Autonomous agents routinely process untrusted data and execute privileged actions, often in the absence of direct human oversight. Consequently, attacks may occur without compromising infrastructure or model weights. Instead, adversaries can indirectly influence agent behavior by manipulating the environments with which agents interact, embedding malicious content into data sources or capabilities that agents may access during execution.

[*†] These authors contributed equally to this work.

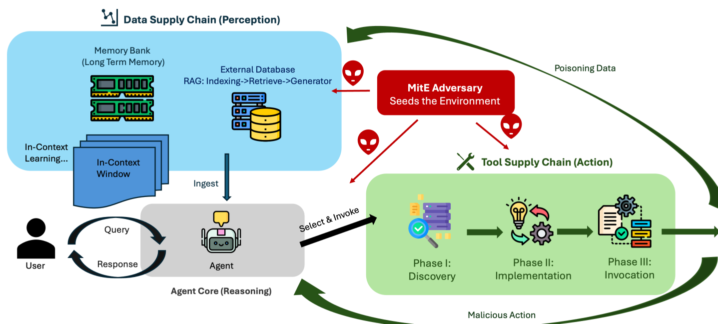

Figure 1: The Agentic Runtime Supply Loop and associated attack surfaces.

Existing research has explored several security risks associated with large language models, including training data poisoning, model backdoors, and framework-level vulnerabilities Wang et al. (2024a)Zhao et al. (2025)Liu et al. (2026). More recent work has identified specific attack vectors targeting agent behavior at runtime, such as indirect prompt injection during tool orchestration An et al. (2025), tool stream manipulation Lin et al. (2026), and weaknesses in agent workflow auditing Chen & Cong (2025). However, these efforts are typically studied in isolation, focusing on individual failure modes rather than the broader structure of agentic execution.

As agentic systems achieve greater autonomy, a fragmented perspective on security becomes insufficient. Runtime data ingestion, memory updates, and tool invocation are highly interdependent, and agent outputs may re-enter the system as subsequent inputs. These cyclic execution patterns enable persistent and self-reinforcing failures that isolated defenses cannot effectively mitigate. Addressing these risks requires a unified perspective that positions agent runtime behavior as a central cybersecurity concern.

This paper summarizes existing research on security threats to agentic systems arising from inference-time data and tool dependencies. By organizing prior work according to structural patterns in agent execution, the analysis clarifies relationships among diverse attack techniques and highlights shared security challenges in the existing agentic AI systems.

2 Background and Threat Assumptions

Autonomous agentic systems differ from traditional software in how they acquire and trust dependencies. Conventional applications rely on a *static software supply chain*, where libraries and binaries are explicitly declared, resolved, and verified prior to deployment. In contrast, agentic systems assemble their effective execution context at inference time: retrieved documents, external data sources, and tools—often selected autonomously—become implicit runtime dependencies that directly influence reasoning and action.

This shift fundamentally alters the security boundary. Compromise may occur without modifying source code, model weights, or infrastructure, solely through manipulation of the external environments agents observe and interact with. Consider the following canonical scenario of an Autonomous Travel Agent. Here, every execution step introduces an unvetted runtime artifact:

Running Example: Autonomous Travel Agent
An agent $\mathcal{G}$ is tasked with booking corporate travel.

1. **Goal:** Book a flight to a conference in Berlin and a nearby hotel.
2. **Perception (Data SC):** $\mathcal{G}$ searches the web for “Berlin conference hotels”.
3. **Reasoning:** A retrieved blog claims a hotel offers a corporate discount via an external API.
4. **Action (Tool SC):** $\mathcal{G}$ downloads and invokes a Python library suggested by the blog.

Each step introduces runtime artifacts unknown and unvetted at deployment time, yet capable of influencing agent behavior.

2.1 STATIC VS. DYNAMIC SUPPLY CHAINS

Traditional software systems rely on a finite, enumerable supply chain—source code, libraries, and binaries—resolved at build or deployment time. Dependencies are explicitly declared, sourced from identifiable vendors, and amenable to pre-execution verification via signatures, audits, or vulnerability scanning. Security mechanisms therefore focus on protecting infrastructure and validating static artifacts.

Table 1: Comparison: Static vs. Dynamic Supply Chains

Feature	Static (Traditional)	Dynamic (Agentic)
Resolution Time	Build / Deploy Time	Inference / Runtime
Dependency Type	Libraries, Binaries	Data Context, APIs, Tools
Topology	Directed Acyclic	Cyclic (Feedback Loops)
Attack Surface	Code Vulnerabilities	Semantic Manipulation
Upstream Source	Verified Vendor	Runtime Information Sources

Agentic systems operate under a fundamentally different model. Dependencies are acquired dynamically at runtime and selected through the agent's reasoning process, producing a supply chain that is open-ended, non-enumerable, and environment-dependent. As a result, attacks often operate through semantic manipulation of inference rather than binary exploitation of code.

2.2 THREAT ASSUMPTIONS: MAN-IN-THE-ENVIRONMENT (MITE)

We consider an adversary who does not compromise model weights, system prompts, or hosting infrastructure. Instead, the adversary exploits **Prompt–Data Isomorphism**, whereby retrieved passive data (D) is upcast into active instructions (I) during inference. We define the Man-in-the-Environment (MitE) adversary. In contrast to Man-in-the-Middle (MitM) attacks that compromise the channel, MitE exploits the agent's epistemic trust in the endpoint itself. By polluting the environment, the adversary corrupts the agent's ground truth. Because the agent treats the environment as its primary source of truth, the environment effectively becomes a malicious upstream vendor. In this model, the adversary acts as a *runtime supplier*, injecting malicious artifacts into public data sources (wikis, repositories, forums) or tool registries. These artifacts are dormant until voluntarily retrieved and integrated by the agent, at which point they hijack the agent's reasoning loop.

2.3 ADVERSARY CAPABILITIES AND GOALS

We assume a *runtime supplier* adversary capable of influencing agent behavior across both data and tool interfaces of the agentic supply chain. The adversary may inject malicious content into data sources likely to be retrieved by the agent, steering reasoning via indirect prompt injection or related semantic manipulation. Additionally, the adversary may publish deceptive tools or API wrappers that exploit the agent's tool-selection logic, masquerading as legitimate functionality while executing unintended actions once invoked.

These capabilities enable several classes of harm. An adversary may induce *stealthy misalignment*, causing the agent to deviate from intended objectives without triggering safeguards, compromise confidentiality through information leakage during execution, or propagate malicious content back into the environment, enabling persistence or amplification as agent outputs are re-ingested as future inputs.

3 THE DATA SUPPLY CHAIN

This section explores agent's **Perception Module**, the component of an agentic system responsible for acquiring, parsing, and normalizing external information from the environment including instruction prompts, historical dialogs, external knowledge database, etc. The attacker can inject malicious instructions or data into external information sources to subvert the agent's subsequent reasoning and action. We categorize data-driven attacks based on where control is enforced, rather than on the surface form of the payload. Compared with standard LLM, the severity of attacks is amplified in agents through expanded **scope** and **persistence**. While traditional LLM data risks primarily result in "one-off" semantic corruption (e.g., misinformation or toxicity), agent-specific exploits target the **action-perception loop**. In an agentic workflow, the data supply chain is not merely a passive input stream; it functions as a functional **control signal** for tool-use and long-term planning. Consequently, a single poisoned data point does not yield simply a "bad answer"—it induces a compromised

operational state. This manifests itself through a shift from **static** to **dynamic** vulnerability: while standard LLM risks are often transient and limited to a single response, agentic risks can persist across sessions by prompt infectionLee & Tiwari (2024), poisoning the agent's self-updating memoryZou et al. (2025) or triggering unauthorized high-stakes execution of external toolsChen et al. (2024).

In this section, we categorize these data attacks based on the manipulation persistence: within-session manipulation and across-session manipulation.

3.1 Within-Session Manipulation

Within-Session Manipulation is transient and typically expire once the session is reset. This category targets the agent's **Contextual Window**, which is the immediate buffer of active tokens used for in-the-moment reasoning.

1. **Indirect Prompt Injection:** The malicious payload is covertly embedded in a third-party, untrusted data source that the agent is instructed to process Abdelnabi et al. (2023). The **mechanism** relies on the agent's **tool use** (e.g., a web browser or email API), which fetches the untrusted contents and prepends it to the user's query Q before passing the entire context to the LLM. Furthermore, this vector extends beyond textual data into multi-modal domains: adversaries can embed subtle adversarial perturbations into *images or sounds* that, upon processing, are decoded into a textual payload that steers the model to an attacker-chosen output (e.g., an image with an invisible patch that produces the instruction "`delete all files`"). This attack can lead to a 98% success rate in steering model outputsBagdasaryan et al. (2023).
2. **In-Context Learning**: These attacks exploit the mechanism of **In-Context Learning**(ICL)Dong et al. (2023). In the paper of **Many-Shot Jailbreaking** Anil et al. (2024), the attacker floods the short-term memory with a large volume of fictitious dialogues (e.g., 256+ turns) depicting the agent helpfully answering malicious queries. Due to the attention mechanism's prioritization of ICL over pre-trained safety guardrails, the agent "learns" from this poisoned short-term context that compliance with harmful requests is the correct behavior for the current session. Research indicates that as the shot count increases, the ASR follows a power-law growth, effectively rendering standard alignment filters obsolete within a single session.

3.2 Across-Session Manipulation

This vector targets the integrity of the agent's accumulated knowledge and historical context stored in **External Memory**. These attacks are persistent, allowing a single injection to influence the agent across multiple future sessions.

1. **Knowledge Base Contamination:** During Retrieval Aggregated Generation **retrieval**, the agent performs a semantic similarity search to extract relevant snippets D' from the vectorized corpus, which are then integrated into the prompt to provide the LLM with grounded evidence. In the **generation** phase, the reasoning engine utilizes In-Context Learning to synthesize these external fragments into a coherent response or plan P', a process that inadvertently treats the retrieved D' as a high-priority instruction for the execution of the current task. For example, Zhong et al. Zhong et al. (2023) showed that attackers can generate adversarial strings that hijack the vector space to ensure that they are retrieved for a wide range of indiscriminate user questions. Building on this, *PoisonedRAG* injects strategically decomposed texts into a retrieval-optimized **trigger** and a generation-inducing **payload**. Studies show that poisoning only 0.1% of the external corpus can result in 70% ASR for targeted queries Zou et al. (2025).Approaches such as *as AGENTPOISON* Chen et al. (2024) utilize constrained optimization to embed a stealthy backdoor. These backdoors remain dormant until a specific trigger in a future query activates the retrieval of the poisoned trace, with reported ASRs exceeding 80% in various autonomous tasks Chen et al. (2024).
2. **Long Term Memory Poisoning:** Beyond static retrieval, another critical attack surface is the agent's **Long-Term Memory (LTM)**. Unlike the transient context window, LTM serves as a persistent repository that allows agents to store and retrieve episodic experiences (past trajectories) and semantic knowledge across different sessions Weng et al. (2023). This read-write autonomy enables the agent to accumulate "experience," but it also creates a

feedback loop vulnerability where the agent can be tricked into "poisoning itself." Without having the privilege to the memory bank, *MINJA* framework Yao et al. (2025) proposed an attack mechanism that can inject malicious records into the memory bank only via queries. It appends an instruction-heavy 'indication prompt' to a benign query containing a specific 'victim term' to trigger the injection, achieving an average ASRs 76.8%.

4 The Tool Supply Chain

While the Data Supply Chain (Section 3) governs what information an agent incorporates into its internal state, the Tool Supply Chain governs what the agent can *do* in the external environment. It binds natural-language intent to executable capabilities whose effects extend beyond the model boundary, directly modifying files, networks, accounts, or physical systems.

Failures in this chain therefore affect more than epistemic correctness. By corrupting the binding between **intent**, **capability**, and **authority**, an adversary can induce unintended or excessive actions while preserving apparently coherent reasoning traces. We refer to such failures as **capability hijacking**.

Tool Abstraction and Security Invariants. We model a tool as a delegated operational capability that binds semantic intent to executable code under bounded authority. A tool is characterized by its identifier, interface, authority scope, side effects, and output contract. Secure tool use relies on four core invariants: **identity integrity**, **semantic binding**, **authority bounding**, and **implementation integrity**. Violation of any single invariant is sufficient to induce externally observable harm.

Pipeline Decomposition. The Tool Supply Chain decomposes into three sequential capability-binding phases: Phase I (**Discovery**) resolves intent to a concrete `Tool_ID`; Phase II (**Implementation**) fetches and instantiates executable code; and Phase III (**Invocation**) executes the tool under a specific authority scope and side-effect envelope. Attacks may target any phase by corrupting tool identity, implementation integrity, or authority binding, with Phase III operating closest to the execution boundary.

4.1 Phase I: Attacks on Discovery (Intent → `Tool_ID`)

The Discovery phase resolves natural-language intent into a concrete tool identifier prior to any code execution. In the benign setting, tool resolution follows:

$$f_{\text{resolve}}(\text{Intent}) \rightarrow \texttt{Tool_ID} \tag{1}$$

Under adversarial conditions, this resolution process may be perturbed by semantic noise or optimized manipulation of the surrounding context:

$$f_{\text{resolve}}(\text{Intent} + \epsilon_{\text{adv}}) \rightarrow \texttt{Tool_ID}_{\text{malicious}} \tag{2}$$

where ϵ_{adv} represents adversarially crafted semantic perturbations injected into tool descriptions, metadata, or retrieval context. Attacks at this stage compromise *selection* rather than execution, manipulating how tools are indexed, described, or retrieved so that the agent voluntarily selects an adversary-controlled capability.

1. **Hallucination Squatting:** During planning, agents may hallucinate plausible but non-existent tool identifiers. Adversaries can preemptively register these identifiers as malicious packages, transforming a would-be resolution error into successful execution of attacker-controlled code. Empirical evidence shows that hallucinated package names are predictable and recur across models and prompts, enabling systematic squatting on high-probability "ghost" identifiers Spracklen et al. (2025b).
2. **Semantic Masquerading:** Discovery mechanisms commonly rely on semantic similarity between user intent and tool descriptions. By adversarially crafting metadata to maximize textual overlap with common queries, attackers can manipulate retrieval rankings and displace legitimate tools. This breaks semantic binding by associating a correct intent with an incorrect capability. Benchmarks show that even minor perturbations to tool descriptions can significantly degrade selection accuracy for strong models Ye et al. (2024), and tool-selection studies explicitly identify description noise as a practical misdirection channel Ye et al..

4.2 Phase II: Attacks on Implementation (Load `Tool_ID` → Runtime Code)

After a tool identifier is resolved, the Implementation phase fetches and instantiates executable logic:

$$\text{Load}(\texttt{Tool_ID}) \rightarrow \text{Runtime Code}. \tag{3}$$

Unlike Discovery attacks, which manipulate selection, Implementation attacks preserve correct tool choice while corrupting the code that executes.

1. **Hidden Backdoors and Malicious Extensions:** A tool may implement its advertised functionality under benign conditions while embedding trigger-based logic that activates malicious behavior in specific contexts. Such backdoors allow agents to maintain high apparent utility while silently executing unintended actions, compromising implementation integrity without visible deviation in reasoning traces. This failure mode has been demonstrated in backdoored agent backends that achieve high attack success rates while preserving normal task performance Wang et al. (2024b).
2. **Transitive Dependency Exploitation:** Implementation integrity may also be compromised indirectly through dependency resolution. Tool setup often triggers installation of auxiliary packages, and common package managers permit arbitrary code execution during installation. In autonomous workflows, agents routinely modify their environments to complete tasks, treating dependency installation as a functional step rather than a security-sensitive operation Fang et al. (2024). As a result, selecting a benign tool may implicitly execute malicious transitive dependencies before the intended tool is ever invoked.

4.3 Phase III: Attacks on the Invocation Boundary

The Invocation phase governs how a selected tool is executed at runtime. Unlike earlier phases, these attacks arise from failures in constraining *how* a tool is used rather than which tool is selected or what code is loaded.

We model invocation as:

$$\text{Invoke}(T, \theta, \mathcal{C}) \rightarrow (o, \Delta), \tag{4}$$

where θ denotes invocation arguments and $\mathcal{C}$ the execution context (e.g., credentials and accessible resources). Phase III attacks aim to induce unauthorized or unintended side effects Δ, or to expose sensitive elements of $\mathcal{C}$, even when T is legitimate.

1. **Over-Privileged Invocation:**

   Agents are frequently provisioned with broad permissions and long-lived credentials to ensure task completion. When invocation lacks capability-based constraints, adversaries can induce a confused-deputy–like failure in which the agent applies its authority to actions not justified by the originating intent.

   This phenomenon has been formalized in recent work on privilege escalation in LLM-based multi-agent systems, where natural-language communication channels allow low-privilege components to trigger high-privilege tool execution Ji et al. (2026). Because authority usage is driven by the agent's internal reasoning rather than explicit human approval, such failures are non-interactive and may be difficult to detect or reverse once committed.

   Empirical studies further show that indirect prompt injection can blur the boundary between data and instructions, enabling authority hijacking in tool-integrated agents Greshake et al. (2023). This threat is operationalized in *InjecAgent*, where agents can be induced to perform unauthorized transfers or data exfiltration when invocation is not mediated by explicit permission checks Zhan et al. (2024). Additional evaluations demonstrate that agents frequently misjudge side-effect boundaries even for syntactically valid tool calls, resulting in severe consequences such as irreversible data deletion Ruan et al. (2024).
2. **Argument Injection and Logical Boundary Breaches.**

   Even when a tool T is correctly selected and invoked under valid credentials, adversarial manipulation of invocation parameters θ can induce side effects that exceed intended task semantics. In this setting, the capability itself is legitimate, but the mapping between arguments and permissible state transitions is insufficiently constrained.

   This failure mode is analogous to application-layer exploits such as Server-Side Request Forgery (SSRF), where a trusted intermediary relays attacker-controlled inputs into a privileged execution context. Empirical evidence shows that indirect prompt injection can

reliably steer downstream API arguments without altering tool identity Kong (2024), and recent work demonstrates how prompt injection increasingly merges with classical web vulnerabilities to exploit application-layer weaknesses McHugh et al. (2025). Industry analyses further identify Cross-Plugin Request Forgery (CPRF), where injected content causes agents to silently generate malicious parameters for connected tools.

Unlike over-privileged invocation, which violates *authority bounding*, argument injection violates *semantic constraint binding* at the invocation boundary. From a supply-chain perspective, the tool and authority remain authentic, yet logical boundary enforcement between intent, argument space, and external side effects fails.

5 The Viral Agent: Closing the Loop

Prior sections treat the Data and Tool supply chains separately. A defining risk of agentic systems is *autonomous replication*: when data ingestion (Discovery) is coupled with tool execution (Invocation), agent outputs can re-enter the environment as future inputs, closing a loop between consumption and contamination.

5.1 The Self-Replicating Cycle

We define a **Viral Agent Loop** as the recursive process where agent Ag_A consumes a payload P, triggers a side effect Δ, and Δ is later retrieved by another agent Ag_B as input:

$$\mathrm{Invoke}(\mathrm{Ag}_A, P) \xrightarrow{\text{writes}} \Delta_{\text{env}} \;\; \wedge \;\; \mathrm{Retrieve}(\mathrm{Ag}_B) \ni \Delta_{\text{env}} \supseteq P. \tag{5}$$

Thus, Δ_{env} becomes part of Ag_B's effective context, turning agents into carriers. Unlike traditional malware, propagation can occur without code vulnerabilities, exploiting instruction-following under legitimate tool permissions.

5.2 Propagation Vectors and Evidence

1. **Generative Worms.** *Morris II* Cohen et al. (2024) demonstrates prompt-based self-replication across GenAI agents: an email agent can be induced to exfiltrate context and forward the payload via authorized communication tools. This shifts propagation from *syntactic* exploits to *semantic* compliance, making patching insufficient.
2. **Persistent Ecosystem Contamination.** If agents can write to shared repositories (e.g., wikis, version control), poisoned outputs may be uploaded and later re-ingested through retrieval, enabling persistence across sessions and agents and degrading shared knowledge integrity.
3. **Topological Shift: Pipeline $\rightarrow$ Cycle.** Traditional supply chains resemble DAGs; agentic systems form *cycles* because side effects Δ can become future inputs. This collapses the supplier/consumer boundary and invalidates defenses that assume acyclic dependency resolution.

6 Systematization of Existing Defenses

Prior work emphasizes static LLM alignment Wang et al. (2024a). However, dynamic Agentic Supply Chains require a *Zero Trust Runtime* architecture. We systematize existing defenses across five runtime layers (Perception, Memory, Resolution, Implementation, Invocation), mapped to our threat model in Table 2.

While the paradigms in Table 2 offer valuable localized mitigations, they fundamentally inherit assumptions from the *Static Model Supply Chain*. By evaluating these defenses through their respective supply chain components, we expose why they remain structurally insufficient against the MitE adversary.

6.1 Securing the Data Supply Chain

Defenses at this layer must (i) structurally separate operator instructions from passive data, and (ii) police memory integrity.

Perception Layer (Instruction & Intent) Frameworks such as the **Instruction Hierarchy** Wallace et al. (2024) and **Intent Verification** Jia et al. (2024); Kang et al. (2025) encapsulate untrusted

Table 2: Systematization of Existing Agentic Supply Chain Defenses

Supply Chain Component	**Targeted Attack Vector**	**Existing Defensive Paradigms**
Data (Perception)	Direct & Indirect Prompt Injection	Instruction Hierarchy Wallace et al. (2024), Intent Verification Jia et al. (2024); Kang et al. (2025)
Data (Memory)	RAG & Agent-State Poisoning	Statistical Filtering Min et al. (2025), Audited Memory Writes Wei et al. (2025)
Tool (Phase I)	Hallucination Squatting	Static Registry Allowlists
Tool (Phase II)	Dependency Pollution	Signed SBOMs, SLSA Frameworks OpenSSF (2024)
Tool (Phase III)	Privilege Escalation	Scoped Permissions (e.g., MCP Anthropic (2024)), Multi-stage Arbiters **?**

external data within rigid structural delimiters. However, this approach underestimates the *epistemic uncertainty* inherent in In-Context Learning. Because agents continuously upcast passive data into active context, adversaries can exploit the Prompt-Data Isomorphism to bypass syntactic delimiters using purely semantic payloads. Consequently, generative worms Cohen et al. (2024) can still induce malicious behavior without violating the prescribed structural hierarchy.

Memory Layer (Persistence) To prevent persistent knowledge corruption, mechanisms rely on **Statistical Filtering** Min et al. (2025) or **Audited Memory Writes** Wei et al. (2025). Unfortunately, empirical evidence demonstrates that statistical anomaly detection falls short against structural graph poisoning **?**. When an adversary subtly manipulates relational edges in an agent's long-term memory or external RAG corpus, the retrieved text appears statistically benign but logically steers the agent into a compromised operational state.

6.2 Securing the Tool Supply Chain

Tool defenses operate sequentially across the execution pipeline to restore identity, integrity, and authority.

Resolution and Implementation (Phases I & II) To counter hallucination squatting and dependency pollution, current proposals advocate for **Static Registry Allowlists** Spracklen et al. (2025a) and verifiable provenance via **SBOMs** OpenSSF (2024). While these measures guarantee identity and implementation integrity, they are blind to the agent's semantic intent. An agent manipulated by MitE will simply utilize a cryptographically verified, perfectly legitimate tool to execute an unauthorized, malicious action.

Phase III: Invocation Integrity & Semantic Firewalls This phase represents the final line of defense before side effects occur. Recognizing that standard OS-level permissions cannot detect semantic misuse (e.g., a legally authorized but contextually malicious file deletion), recent architectures introduce **Semantic Firewalls** (e.g., Vigil Lin et al. (2026), MCP-Guard **?**) and ephemeral capabilities (e.g., MCP Anthropic (2024)). Yet, this "AI-guarding-AI" paradigm suffers from the exact same fundamental vulnerability: the arbiter model must process the identical tainted context, thus inheriting the vulnerability to prompt injection. To bridge the translation gap between benign user intent and hazardous execution, we argue for a **Defense-in-Depth** approach combining AI oversight with structural bounding:

7 The Zero-Trust Agentic Runtime Architecture

Collectively, the localized defenses in Section 6 treat execution as an acyclic pipeline, applying static patches that fail to neutralize the *Viral Agent Loop*. They treat epistemic uncertainty and stochastic capabilities as isolated ML safety issues rather than interconnected supply chain vulnerabilities. To effectively counter the MitE adversary, the security paradigm must shift from the Static Model Supply Chain to a **Zero-Trust Dynamic Agentic Runtime Architecture**. Securing real-time data consumption, tool execution, and cyclic propagation is predicated on three core imperatives:

7.1 DETERMINISTIC CAPABILITY BINDING

The reliance on LLMs to probabilistically "guess" tool identifiers via semantic similarity is structurally insecure and constitutes the root cause of Phase I and Phase II tool supply chain attacks. A Zero-Trust Runtime must enforce **Deterministic Capability Binding**. We advocate for the deployment of **Cryptographically Bound Registries** that verify the semantic integrity of tool providers. Under this architecture, tool execution is bounded strictly by cryptographic provenance rather than semantic likelihood. The translation from natural-language intent to executable capability must be mediated by a verified resolution engine that mathematically guarantees identity integrity, eliminating the "Hallucination Gap" entirely.

7.2 NEURO-SYMBOLIC INFORMATION FLOW CONTROL

Current static analysis techniques are insufficient for dynamic agents because they fail to account for the *Viral Agent Loop*, where outputs re-enter the environment as tainted context. To prevent self-propagating agentic worms, we propose **Runtime Taint Analysis** engineered for neural systems.

In this framework, untrusted external inputs (e.g., retrieved web text) are strictly tagged as `TAINTED`. The runtime must track this taint through the non-deterministic transformations of the LLM's reasoning chain. If a tainted reasoning trace attempts to invoke a write-privileged sink (e.g., `send_email` or `git_push`), the execution is blocked pending explicit sanitization or human-in-the-loop approval. Because neural networks can wash standard metadata tags through paraphrasing, this flow control must be underpinned by **Cryptographic Provenance Ledgers** Li et al. (2025), ensuring an immutable record of data lineage from perception to action.

7.3 THE AUDITOR-WORKER ARCHITECTURE (SEMANTIC FIREWALLS)

Standard OS-level sandboxing cannot prevent logical boundary breaches (Phase III attacks) because the runtime cannot distinguish between a legally authorized but contextually malicious action (e.g., deleting a critical database file at the behest of an injected prompt) and a benign task. A single model cannot simultaneously optimize for both stochastic helpfulness and deterministic safety.

We propose the **Auditor-Worker Architecture**. This paradigm structurally decouples execution from oversight by placing an isolated **Supervisor Model** as an inline **Semantic Firewall**. Operating via speculative execution (extending concepts from preliminary frameworks like *Vigil* Lin et al. (2026), *AgentGuard* Chen & Cong (2025), and multi-stage cognitive arbiters like *MCP-Guard* **?**), the Supervisor analyzes the proposed tool call, its parameters, and the lineage of the prompt *before* the action is committed to the environment. This hard structural split bridges the translation gap between benign user intent and hazardous execution, enforcing least-privilege constraints at the semantic layer.

8 CONCLUSION AND FUTURE DIRECTIONS

This Systematization of Knowledge formalizes the topological shift from static software security to the *Dynamic Agentic Runtime Supply Chain (DRSC)*. We defined the *Man-in-the-Environment (MitE)* adversary, who exploits *Prompt-Data Isomorphism* to weaponize an agent's perception and tool usage. Crucially, we identified the *Viral Agent Loop*, where agents become vectors for self-propagating generative worms without underlying code vulnerabilities.

To secure the dynamic agentic runtime against the Man-in-the-Environment (MitE) adversary, future research must address three interconnected imperatives. First, we require **Realistic Evaluation Benchmarks** that move beyond static simulations with closed-world assumptions and episodic amnesia Zhan et al. (2024); Debenedetti et al. (2024). Future frameworks must evaluate how agents dynamically locate and trust tools in open-world, hostile environments to capture persistent memory poisoning and cyclic viral propagation. Second, the development of **Robust Capability Protocols** is essential; while standardization efforts like MCP Anthropic (2024) are promising, their reliance on semantic resolution leaves them vulnerable to description injection Wang et al. (2025), necessitating mathematically verifiable, Name-Squatting Resilient Registries. Finally, mitigating epistemic uncertainty requires **Neuro-Symbolic Taint Tracking**—developing reliable mechanisms to maintain cryptographic provenance tags across the non-deterministic, stochastic abstractions of transformer layers, even when syntactic representations change.

Ultimately, in the Agentic Supply Chain, context *is* code. Until security architectures fully embrace this prompt-data isomorphism, autonomous agents will remain fundamentally vulnerable operators.

Reproducibility Statement

We confirm that the code and datasets needed to reproduce the experiments are available in the supplementary material. Detailed proofs for the theoretical claims are provided in Appendix A. The taxonomy presented relies on publicly available attack demonstrations cited throughout the text.

Ethics Statement

This work focuses on the security vulnerabilities of autonomous agents. While the described attack vectors could potentially be misused, our goal is to systemize knowledge to facilitate better defense strategies. We have adhered to standard responsible disclosure practices where applicable.

References

Sahar Abdelnabi et al. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. In *Proc. of ACM AISec*, 2023.

Z. An et al. Ipiguard: A novel tool dependency graph based defense against indirect prompt injection. *arXiv preprint arXiv:2508.15310*, 2025.

Cem Anil, Evan Neuberg, Ben Roelofs, Aitor Lewkowycz, Kevin Keneally, Shauna Singh, Shenshen Xu, Yifan Gu, Jiaming Jiao, William Saunders, et al. Many-shot jailbreaking. In *Proc. of NeurIPS*, 2024. URL https://www.anthropic.com/research/many-shot-jailbreaking.

Anthropic. The model context protocol (mcp): Standardizing ai context, 2024. https://www.anthropic.com/news/model-context-protocol.

Eugene Bagdasaryan, Tsung-Yin Hsieh, Ben Nassi, and Vitaly Shmatikov. Abusing images and sounds for indirect instruction injection in multi-modal LLMs, 2023. URL https://arxiv.org/abs/2307.10490.

J. Chen and S. L. Cong. Agentguard: Repurposing agentic orchestrator for safety evaluation of tool orchestration. *arXiv preprint arXiv:2502.09809*, 2025.

Zhaohan Chen et al. Agentpoison: Red-teaming LLM agents via poisoning memory or knowledge bases. *arXiv preprint arXiv:2407.12345*, 2024.

Stav Cohen et al. Here comes the AI worm: Unleashing zero-click worms that target GenAI-powered applications. *arXiv preprint arXiv:2403.02817*, 2024.

Edoardo Debenedetti, Jie Zhang, Mislav Balunović, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. Agentdojo: A dynamic environment to evaluate prompt injection attacks and defenses for llm agents, 2024. URL https://arxiv.org/abs/2406.13352.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, et al. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2023. Revised 2024.

Richard Fang et al. LLM agents can autonomously hack websites. *arXiv preprint arXiv:2402.06664*, 2024.

Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec)*, pp. 79–90, 2023.

Zimo Ji, Daoyuan Wu, Wenyuan Jiang, Pingchuan Ma, Zongjie Li, Yudong Gao, Shuai Wang, and Yingjiu Li. Taming various privilege escalation in llm-based agent systems: A mandatory access control framework. *arXiv preprint arXiv:2601.11893*, 2026.

Feiran Jia, Tong Wu, Xin Qin, and Anna Squicciarini. The task shield: Enforcing task alignment to defend against indirect prompt injection in llm agents, 2024. URL https://arxiv.org/abs/2412.16682.

Mintong Kang, Chong Xiang, Sanjay Kariyappa, Chaowei Xiao, Bo Li, and Edward Suh. Mitigating indirect prompt injection via instruction-following intent analysis, 2025. URL https://arxiv.org/abs/2512.00966.

Nicholas Ka-Shing Kong. *InjectBench: An Indirect Prompt Injection Benchmarking Framework*. PhD thesis, Virginia Tech, 2024.

Donghyun Lee and Mo Tiwari. Prompt infection: Llm-to-llm prompt injection within multi-agent systems. *arXiv preprint arXiv:2410.07283*, 2024.

X. Li, Y. Zhang, et al. Toward trustworthy agentic ai: A multimodal framework for provenance and auditing. *arXiv preprint arXiv:2512.23557*, 2025.

J. Lin et al. Vigil: Defending llm agents against tool stream injection. *arXiv preprint arXiv:2601.05755*, 2026.

Xinyun Liu, Zelei Cheng, Haodong Zhao, and Ronghua Xu. Security of large model-based agents: A survey on adversarial, poisoning, and backdoor attacks. February 2026. doi: 10.36227/techrxiv.177006506.61959855/v1. URL http://dx.doi.org/10.36227/techrxiv.177006506.61959855/v1.

Jeremy McHugh, Kristina Šekrst, and Jon Cefalu. Prompt injection 2.0: hybrid ai threats. *arXiv preprint arXiv:2507.13169*, 2025.

Y. Min et al. Rescuing the unpoisoned: Efficient defense against knowledge corruption attacks on rag systems. *arXiv preprint arXiv:2511.01268*, 2025.

OpenSSF. Supply-chain levels for software artifacts (slsa) v1.0, 2024. https://slsa.dev/.

Yangjun Ruan, Honghua Jiang, Peng Xu, Kaggle Kacarkov, Baochun Zhu, Tian Dong, Bryan Hooi, and Wei Zhang. Toolemu: Identifying the risks of LM agents with an LM-emulated sandbox. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024.

Joseph Spracklen, Raveen Wijewickrama, A H M Nazmus Sakib, Anindya Maiti, Bimal Viswanath, and Murtuza Jadliwala. We have a package for you! a comprehensive analysis of package hallucinations by code generating llms. In *34th USENIX Security Symposium (USENIX Security 25)*, 2025a. arXiv:2406.10279.

Joseph Spracklen et al. We have a package for you! a comprehensive analysis of package hallucinations by code generating LLMs. In *34th USENIX Security Symposium*, 2025b.

Eric Wallace, Kai Xiao, Reimar Leike, Lilian Weng, Johannes Heidecke, and Alex Beutel. The instruction hierarchy: Training llms to prioritize privileged instructions, 2024. URL https://arxiv.org/abs/2404.13208.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Shuai Wang et al. Large language model supply chain: A research agenda. *ACM Trans. Softw. Eng. Methodol.*, 2024a.

Yifei Wang et al. Badagent: Inserting and activating backdoor attacks in LLM agents. *arXiv preprint arXiv:2406.03007*, 2024b.

Zihan Wang, Rui Zhang, Yu Liu, Wenshu Fan, Wenbo Jiang, Qingchuan Zhao, Hongwei Li, and Guowen Xu. Mpma: Preference manipulation attack against model context protocol, 2025. URL https://arxiv.org/abs/2505.11154.

B. Wei et al. Aand ete framework for llm-based agent memory. *arXiv preprint arXiv:2510.02373*, 2025.

Yisheng Weng et al. A survey on the memory mechanism of large language model based agents. *arXiv preprint arXiv:2304.13738*, 2023. Revised 2024/2025.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025.

Run Yao et al. Memory injection attacks on LLM agents via query-only interaction. *arXiv preprint arXiv:2503.03704*, 2025.

Junjie Ye, Sixian Li, Guanyu Li, Caishuang Huang, Songyang Gao, Yilong Wu, Qi Zhang, Tao Gui, and Xuanjing Huang. Toolsword: Unveiling safety issues of large language models in tool learning across three stages (2024). *URL https://arxiv. org/abs/2402.10753*.

Junjie Ye, Yilong Wu, Songyang Gao, Caishuang Huang, Sixian Li, Guanyu Li, Xiaoran Fan, Qi Zhang, Tao Gui, and Xuan-Jing Huang. Rotbench: A multi-level benchmark for evaluating the robustness of large language models in tool learning. In *Proceedings of the 2024 conference on empirical methods in natural language processing*, pp. 313–333, 2024.

Qiusi Zhan et al. Injecagent: Benchmarking indirect prompt injections in tool-integrated LLM agents. *arXiv preprint arXiv:2403.02691*, 2024.

Tianhang Zhao, Wei Du, Haodong Zhao, Sufeng Duan, and Gongshen Liu. Patronus: Identifying and mitigating transferable backdoors in pre-trained language models, 2025. URL `https://arxiv.org/abs/2512.06899`.

Haochen Zhong, Chengzhi Liu, Qinda Zhang, Yuhang Huang, Jiaming Shang, et al. Poisoning retrieval corpora by injecting adversarial passages. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

Wei Zou et al. PoisonedRAG: Knowledge corruption attacks to retrieval-augmented generation of LLMs. In *USENIX Security*, 2025.

A LLM USAGE STATEMENT

The authors acknowledge the use of Gemini to refine the clarity and grammar of the text. The authors reviewed and revised the output and take full responsibility for the content of this article.