

AgentSight: System-Level Observability for AI Agents Using eBPF

Yusheng Zheng
UC Santa Cruz
Santa Cruz, CA, USA
yzhen165@ucsc.edu

Tong Yu
eunomia-bpf Community
China
yt.xyxx@gmail.com

Yanpeng Hu
ShanghaiTech University
Shanghai, China
huyp@shanghaitech.edu.cn

Andi Quinn
UC Santa Cruz
Santa Cruz, CA, USA
aquinn1@ucsc.edu

Abstract

Modern software infrastructure increasingly relies on LLM agents for development and maintenance, such as Claude Code and Gemini-cli. However, these AI agents differ fundamentally from traditional deterministic software, posing a significant challenge to conventional monitoring and debugging. This creates a critical semantic gap: existing tools observe either an agent’s high-level intent (via LLM prompts) or its low-level actions (e.g., system calls), but cannot correlate these two views. This blindness makes it difficult to distinguish between benign operations, malicious attacks, and costly failures. We introduce AgentSight, an AgentOps observability framework that bridges this semantic gap using a hybrid approach. Our approach, *boundary tracing*, monitors agents from outside their application code at stable system interfaces using eBPF. AgentSight intercepts TLS-encrypted LLM traffic to extract semantic intent, monitors kernel events to observe system-wide effects, and causally correlates these two streams across process boundaries using a real-time engine and secondary LLM analysis. This instrumentation-free technique is framework-agnostic, resilient to rapid API changes, and incurs less than 3% performance overhead. Our evaluation shows AgentSight detects prompt injection attacks, identifies resource-wasting reasoning loops, and reveals hidden coordination bottlenecks in multi-agent systems. AgentSight is released as an open-source project at <https://github.com/eunomia-bpf/agentsight>.

1 Introduction

The role of machine learning in systems is undergoing a fundamental shift from optimizing well-defined tasks, such as database query planning, to a new paradigm of *agentic computing*. From a systems perspective, an AI agent couples a Large Language Model’s (LLM) reasoning with direct access to system tools, granting it agency to perform operations like spawning processes, modifying the filesystem, and executing commands. This technology is being rapidly integrated

into production environments, powering autonomous developer tools like Claude Code[4], Cursor Agent[5] and Gemini-CLI[24], which can independently handle complex software engineering and system maintenance tasks. In essence, we are deploying non-deterministic ML systems, creating an unprecedented class of challenges for system reliability, security, and verification.

This paradigm shift creates a critical semantic gap: the chasm between an agent’s high-level *intent* and its low-level *system actions*. Unlike traditional programs with predictable execution paths, agents use LLMs and autonomous tools to dynamically generate code and spawn arbitrary subprocesses. This makes it hard for existing observability tools to distinguish benign operations from catastrophic failures. Consider an agent tasked with code refactoring that, due to a malicious prompt it reads from external url in the search result when search for API documents, instead injects a backdoor (indirect prompt injection)[30]. An application-level monitor might see a successful "execute script" tool call, while a system monitor sees a bash process writing to a file. Neither can bridge the gap to understand that a benign intention has been twisted into a malicious action, rendering them effectively blind.

Current approaches are trapped on one side of this semantic gap. *Application-level instrumentation*, found in frameworks like LangChain [8] and AutoGen [29], captures an agent’s reasoning and tool selection. While these tools see the *intent*, they are brittle, require constant API updates, and are easily bypassed: a single shell command escapes their view, breaking the chain of visibility under a flawed trust model. Conversely, *generic system-level monitoring* sees the *actions*, tracking every system call and file access. However, it lacks all semantic context. To such a tool, an agent writing a data analysis script is indistinguishable from a compromised agent writing a malicious payload. Without understanding the preceding LLM instructions, the *why* behind the *what*, its stream of low-level events is meaningless noise.

We propose boundary tracing as a novel observability method designed specifically to bridge this semantic gap.

Our key insight is that while agent internals and frameworks are volatile, the interfaces through which they interact with the world (the kernel for system operations and the network for communication) are stable and unavoidable. By monitoring from outside the application at these boundaries, we can capture an agent’s high-level intent and its low-level system effects. We present **AgentSight**, a system that realizes boundary tracing using eBPF to intercept TLS-encrypted LLM traffic for intent and monitor kernel events for effects. Its core is a novel, two-stage correlation process: a real-time engine links an LLM response to the system behavior it triggers, and a secondary "observer" LLM performs a deep semantic analysis on the resulting trace to infer risks and explain *why* a sequence of events is suspicious. This instrumentation-free, framework-agnostic technique incurs less than 3% overhead and effectively detects prompt injection attacks, resource-wasting reasoning loops, and multi-agent system bottlenecks.

In summary, our contributions are:

1. We introduce boundary tracing as a principled approach to AI agent observability that bridges the semantic gap by monitoring at stable system interfaces.
2. We present a novel engine that combines real-time, eBPF-based signal matching with LLM-based semantic analysis to provide deep, contextual understanding of agent behavior.
3. We demonstrate AgentSight’s effectiveness in detecting prompt injection attacks, reasoning loops, and multi-agent coordination failures with sub-3% overhead.

2 Background and Related Work

This section outlines LLM agent architecture, reviews existing observability work to highlight the semantic gap, and introduces eBPF as our foundational technology.

2.1 LLM Agent Architecture

The agentic systems described in the introduction are typically implemented using a common architecture. These systems consist of three core components: (1) an LLM backend for reasoning, (2) a tool execution framework for system interactions, and (3) a control loop that orchestrates prompts, tool calls, and state management. Popular frameworks such as LangChain [8], AutoGen [29], Cursor Agent[5], Gemini-CLI[24] and Claude Code[4] all implement variations of this model. This architecture is what enables agents to dynamically construct and execute complex plans (e.g., autonomously writing and running a script to analyze a dataset) based on high-level natural language objectives. Multi-agent LLM systems have also emerged for software development, collaborative task-solving, simulating social behaviors in virtual worlds, and other tasks [15, 28].

2.2 Observability for LLM Agent

Existing approaches are siloed on one side of the semantic gap. Intent-side observability, supported by industry tools like Langfuse, LangSmith, and Datadog [10, 16, 18, 19, 21] and is unifying by standards from the OpenTelemetry GenAI working group [7, 20] and academics conceptual taxonomies [11, 23] under the AgentOps concept, excels at tracing application-level events but is fundamentally blind to out-of-process system *actions*. Conversely, action-side observability with tools like Falco and Tracee [6, 27] offers comprehensive visibility into system calls but lacks the semantic context to understand an agent’s *intent*, failing to distinguish a benign task from a malicious one. A parallel line of research into reasoning-level and interpretability aims to make the agent’s internal thought processes more transparent by reconstructing cognitive traces [26] or enabling explanatory dialogues [17], but these work mainly focus on the llm itself, does not bridge the gap between the agent’s internal reasoning and its external, low-level effects on the system.

2.3 extended Berkeley Packet Filter (eBPF)

To bridge the semantic gap, our approach requires a technology that can safely and efficiently observe both network communication and kernel activity. eBPF (extended Berkeley Packet Filter) is a fundamental advancement in kernel programmability that provides precisely this capability [14]. Originally designed for packet filtering, eBPF has evolved into a general-purpose, in-kernel virtual machine that powers modern observability and security tools [12, 25], and not limited to Linux[22, 32]. For AI agent observability, eBPF is uniquely suited because it allows observation at the exact boundaries where agents interact with the world, enabling both TLS interception for semantic *intent* and syscall monitoring for system *actions* with minimal overhead. Critically, its kernel-enforced safety guarantees, including verified termination and memory safety, make it ideal for production environments and provide a stable foundation for our solution [9].

3 Design

The design of AgentSight is guided by a single imperative: to bridge the semantic gap between an agent’s intent and its actions. We achieve this through a novel observability method, boundary tracing, realized by a multi-signal correlation engine.

3.1 Challenges

The emergent and non-deterministic nature of AI agents fundamentally breaks traditional program observability paradigms, introducing two core challenges that original observability cannot address.

Bridging the Semantic Gap Between Intent and Action The first and most significant challenge is bridging the vast semantic gap between an agent’s high-level *intent* and its low-level system *actions*. Unlike conventional software, where intent is encoded in predictable source code, an agent’s intent is expressed in natural language and interpreted by an LLM, creating dynamic "source code" that is generated at runtime. Consequently, it is impossible for a static analyzer to determine what an agent will do. For example, the intent "find and fix the bug in the authentication module" is semantically rich but operationally ambiguous, potentially resulting in a complex sequence of actions like reading files (`openat2`), compiling code (`execve -> gcc`), and running tests (`execve -> python`). This creates a critical observability problem: how can a monitoring system verify that the cascade of system calls is a legitimate fulfillment of the natural language intent? To solve this, an observer must move beyond simple pattern matching to gain a semantic understanding of the agent’s goal, necessitating a new, llm based approach to interpret the correlated traces.

Isolating the Causal Signal from High-Volume System Noise The second challenge stems from the agent’s autonomy to use any tool necessary to achieve its goal, leading to an unpredictable and high-volume stream of system events. An agent might spawn shells, download scripts, or invoke compilers-processes that are not known ahead of time. This makes it exceedingly difficult to distinguish the agent’s specific activity (the "signal") from the background noise of the operating system. Static, pre-configured filters, for instance, a rule to only monitor `git` commands, are inherently brittle and will fail the moment the agent uses `curl` and `bash` to achieve a similar outcome. Our design addresses this with an aggressive, dynamic in-kernel eBPF filter. By tracking process creation events (`fork`, `execve`), the filter builds a complete lineage tree of the agent’s activity and dynamically applies rules in the kernel to only pass events from the agent or its descendants to userspace. This approach ensures that the entire causal chain is captured efficiently at the source, dramatically reducing overhead and providing a clean, high-fidelity signal for correlation and analysis.

3.2 Boundary Tracing: A Principled Approach

Our key insight is that all agent interactions must traverse well-defined and stable system boundaries: the kernel for system operations and the network for external communications with LLM serving backends (Figure 1). By monitoring at these boundaries rather than within volatile agent code, we achieve comprehensive monitoring independent of implementation details. This approach enables Semantic Correlation, the ability to causally link high-level intentions with low-level system events. This is supported by two principles. First is Comprehensiveness, as kernel-level monitoring ensures no system action from process creation to file I/O goes unobserved, even across spawned subprocesses. Second

is Stability, since system call ABIs and network protocols evolve far more slowly than agent frameworks, providing a durable, future-proof solution. This paradigm shifts the trust model from assuming a cooperative agent to enforcing observation at tamper-proof boundaries.

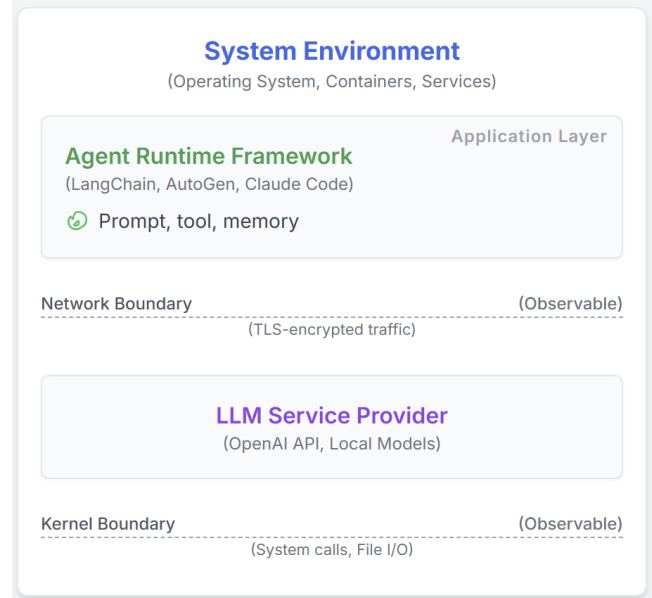


Figure 1. Agent Framework Overview

3.3 System Architecture: Observing the Boundaries

AgentSight’s architecture simultaneously taps into the two critical boundaries. As shown in Figure 2, we use eBPF to place non-intrusive probes that capture a decrypted Intent Stream (LLM prompts/responses) from userspace SSL functions and an Action Stream (syscalls, process events) from the kernel. A userspace correlation engine then processes and joins these streams into a unified, causally-linked trace.

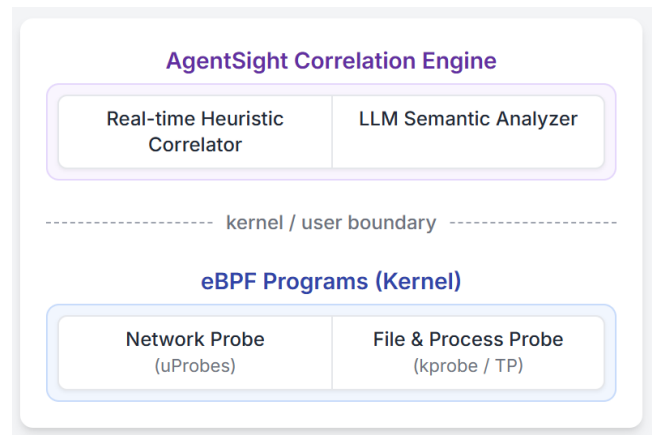


Figure 2. AgentSight System Architecture.

Several key components enable AgentSight to effectively bridge the semantic gap:

eBPF for Safe, Unified Probing: We chose eBPF for its production safety, high performance, and unified ability to access both userspace and kernel data streams. Our design intercepts decrypted data from the agent’s interaction with LLM serving backend, which is more efficient and manageable than network-level packet capture or proxy-based solutions.

Multi-Signal Causal Correlation Engine: The core of our design is a correlation strategy that establishes causality between intent and action. We designed a multi-signal engine that relies on three key mechanisms: Process Lineage, which builds a complete process tree by tracking fork and execve events to link actions in child processes back to the parent agent; Temporal Proximity, which associates actions that occur within a narrow time window immediately following an LLM response; and Argument Matching, which directly matches content from LLM responses, such as filenames, URLs, or commands, with the arguments of subsequent system calls. Together, these signals enable AgentSight to definitively establish causal relationships between high-level intentions and low-level system operations across process boundaries.

LLM-Powered Semantic Analysis: To move beyond brittle, rule-based detection, we designed the system to use a secondary LLM as a reasoning engine. By prompting a powerful model with the correlated event trace, we leverage its ability to understand semantic nuance, infer causality in complex scenarios, and summarize findings in natural language. This "AI to watch AI" approach allows AgentSight to detect threats that do not match predefined patterns.

4 Implementation

AgentSight is implemented as a userspace daemon (6000 lines of Rust/C) orchestrating eBPF programs, with a TypeScript frontend (3000 lines) for analysis. It is designed for high performance, processing raw kernel event streams into correlated, human-readable data.

4.1 Data Collection at the Boundaries

Our eBPF probes capture the raw intent and action streams from the system. To capture semantic intent, an eBPF program with uprobes attaches to `SSL_read/SSL_write` in crypto libraries like OpenSSL to intercept decrypted LLM communications. Our userspace daemon implements a stateful reassembly mechanism to handle streaming protocols such as Server-Sent Events (SSE). To capture system actions, a second eBPF program uses stable tracepoints like `sched_process_exec` to build a process tree and kprobes to dynamically monitor relevant syscalls such as `openat2`, `connect`, and `execve`. To manage the high volume of kernel events without data loss, aggressive in-kernel filtering is applied to

ensure only events from targeted agent processes are sent to userspace, minimizing overhead.

4.2 The Hybrid Correlation Engine

The Rust-based userspace daemon houses our two-stage correlation engine. The first stage consumes events from eBPF ring buffers and performs real-time heuristic linking. This streaming pipeline enriches raw events with context like mapping a file descriptor to a full path, maintains a stateful process tree, and applies the causal linking logic described in our design, using a 100-500ms window for temporal correlation. Once a coherent trace is constructed, the second stage formats it into a structured log for semantic analysis. This log is used to construct a detailed prompt for a secondary LLM, instructing it to act as a security analyst. The LLM’s natural language analysis and confidence score become the final output of our system. A key challenge at this stage is managing the latency and cost of LLM analysis, which our system mitigates through asynchronous processing and robust prompt engineering.

5 Evaluation

Our evaluation is guided by two research questions: First, what is the performance overhead of AgentSight in realistic workflows? Second, how effectively does it bridge the semantic gap to detect critical security threats and performance pathologies, while also revealing complex dynamics in multi-agent systems?

5.1 Performance Evaluation

Table 1. Overhead Introduced by AgentSight

Task	Baseline (s)	AgentSight (s)	Overhead
Understand Repo	127.98	132.33	3.4%
Code Writing	22.54	23.64	4.9%
Repo Compilation	92.40	92.72	0.4%

We evaluated AgentSight on a server (Ubuntu 22.04, Linux 6.14.0) using Claude Code 1.0.62[4] with claude 4 as the test agent. The benchmarks focused on three real-world developer workflows using a tutorial repo[13]: repository understanding with the `/init` command, code generation for `bpfftrace` scripts, and full repository compilation with parallel builds. Each experiment was run 3 times with and without AgentSight to measure runtime overhead. Table 1 quantifies the runtime overhead of AgentSight across three developer workflows, with a average 2.9% overhead.

5.2 Case Studies

We evaluated AgentSight’s effectiveness through case studies that demonstrate its ability to detect security threats, identify

performance issues, and provide insights into complex multi-agent systems.

5.2.1 Case Study 1: Detecting Prompt Injection Attacks. We tested AgentSight’s ability to detect indirect prompt injection attacks[30]. In our test, a software development agent is instructed to clone and build a C project. The project’s README file directed the agent to a URL serving HTML with a hidden prompt. This prompt made the agent read and send `/etc/passwd` to a collection server, using a command cleverly disguised as a necessary step in the build process. AgentSight captured the full correlated attack chain: from the initial URL fetch to the final network exfiltration, with 521 events and merged into 37 events by Correlation Engine. The observer LLM analyzed this trace, returned a high-confidence attack score, and concluded that the agent’s actions were logically inconsistent with its stated goal. This result demonstrates how causally linking intent to system-level actions provides effective, context-aware threat detection.

5.2.2 Case Study 2: Reasoning Loop Detection. An agent attempting a complex task may enter an infinite loop due to a common tool usage error[31]. We implement a research agent using `crewai`[2] with `gpt-4o-mini`[1], it repeatedly called a web search tool with incorrect arguments, received an error, but then failed to correct its mistake, retrying the exact same failing command. AgentSight’s real-time monitors detect this anomalous resource consumption from a trace of API calls and passed it to the observer LLM. The LLM identified the root cause as a persistent tool error, noting the agent was caught in a "try-fail-re-reason" loop; it executed the same failing command, passed the identical error back to the reasoning LLM, and failed to learn from the tool’s output.

5.2.3 Case Study 3: Multi-Agent Coordination Monitoring. AgentSight monitored a team of 6 collaborating software development agents for our Github repo, using `claude-code` subagents[3], captured 3153 total events after Correlation Engine. For instance, frontend agent and test agent sometimes are blocked by sequential dependencies and numerous retry cycles caused by file locking contention during parallel development and testing tasks. The analysis demonstrated that while the agents developed some emergent coordination, separating the roles more clear could reduce total runtime and token cost. This reveals how boundary tracing uniquely captures multi-agent system dynamics that application-level monitoring cannot observe across process boundaries.

6 Conclusion

This paper introduced AgentSight to bridge the critical semantic gap between an AI agent’s intent and its system-level

actions using novel *boundary tracing* approach. By leveraging eBPF, the system monitors network and kernel events without instrumentation, causally linking LLM communications to their system-wide effects via a hybrid correlation engine. Our evaluation shows AgentSight effectively detects prompt injection attacks, reasoning loops, and multi-agent bottlenecks with under 3% performance overhead. This "AI to watch AI" provides a foundational methodology for the secure and reliable deployment of increasingly autonomous AI systems.

References

- [1] 2024. GPT-4o Mini: Advancing Cost-Efficient Intelligence. <https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/>. Blog post announcing GPT-4o mini.
- [2] 2025. crewAI: Framework for Orchestrating Role Playing Autonomous AI Agents. <https://github.com/crewAIInc/crewAI>. GitHub repository, accessed 30 Jul 2025.
- [3] 2025. Subagents. <https://docs.anthropic.com/en/docs/claude-code/sub-agents>. Claude Code documentation page describing custom AI sub agents.
- [4] Anthropic. 2025. Introducing Claude Code. <https://www.anthropic.com/news/claude-code>. Agentic coding tool announcement, Anthropic blog.
- [5] Anysphere Inc. 2025. Cursor: The AI powered Code Editor. <https://cursor.com/>. AI assisted IDE with agent mode; latest release version 1.0 on June 4, 2025.
- [6] The Falco Authors. 2023. Falco: Cloud Native Runtime Security. <https://falco.org/>
- [7] Alexandr Bandurchin. [n. d.]. AI Agent Observability Explained: Key Concepts and Standards. <https://uptrace.dev/blog/ai-agent-observability>. Uptrace Blog, April 16, 2025.
- [8] Harrison Chase. 2023. LangChain: Building applications with LLMs through composability. <https://github.com/langchain-ai/langchain>
- [9] Linux Kernel Community. 2023. BPF Documentation - The Linux Kernel. <https://www.kernel.org/doc/html/latest/bpf/>
- [10] Datadog Inc. [n. d.]. Monitor, troubleshoot, and improve AI agents with Datadog. <https://www.datadoghq.com/blog/monitor-ai-agents/>. Datadog Blog, 2023.
- [11] Liming Dong, Qinghua Lu, and Liming Zhu. 2024. AgentOps: Enabling Observability of LLM Agents. *arXiv preprint arXiv:2411.05285* (2024).
- [12] eBPF Community. 2023. eBPF Documentation. <https://ebpf.io/>
- [13] eunomia-bpf. 2024. eBPF Developer Tutorial. <https://github.com/eunomia-bpf/bpf-developer-tutorial>. Accessed: 2025-01-29.
- [14] Brendan Gregg. 2019. *BPF Performance Tools*. Addison-Wesley Professional.
- [15] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large Language Model based Multi-Agents: A Survey of Progress and Challenges. *arXiv preprint arXiv:2402.01680* (2024).
- [16] Helicone. 2023. Helicone / LLM-Observability for Developers. <https://www.helicone.ai/>
- [17] Been Kim, John Hewitt, Neel Nanda, Noah Fiedel, and Oyvind Tafjord. 2025. Because we have LLMs, we Can and Should Pursue Agentic Interpretability. *arXiv preprint arXiv:2506.12152* (2025).
- [18] LangChain. 2023. Observability Quick Start - LangSmith - LangChain. <https://docs.smith.langchain.com/observability>
- [19] Langfuse. 2024. Langfuse - LLM Observability & Application Tracing. <https://langfuse.com/>
- [20] Guangya Liu and Sujay Solomon. [n. d.]. AI Agent Observability – Evolving Standards and Best Practices. <https://opentelemetry.io/blog/2025/ai-agent-observability/>. OpenTelemetry Blog, March 6, 2025.

- [21] Jannik Maierhöfer. 2025. AI Agent Observability with Langfuse. <https://langfuse.com/blog/2024-07-ai-agent-observability-with-langfuse>. Langfuse Blog, March 16, 2025.
- [22] microsoft. [n. d.]. eBPF for Windows. <https://github.com/microsoft/ebpf-for-windows>.
- [23] Dany Moshkovich and Sergey Zeltyn. 2025. Taming Uncertainty via Automation: Observing, Analyzing, and Optimizing Agentic AI Systems. *arXiv preprint arXiv:2507.11277* (2025).
- [24] Taylor Mullen and Ryan J. Salva. 2025. Gemini CLI: Your Open Source AI Agent. <https://blog.google/technology/developers/introducing-gemini-cli-open-source-ai-agent/>. Google Developers Blog, Jun 2025.
- [25] Cilium Project. 2023. eBPF-based Networking, Observability, and Security. <https://cilium.io/>
- [26] Benjamin Rombaut, Sogol Masoumzadeh, Kirill Vasilevski, Dayi Lin, and Ahmed E. Hassan. 2025. Watson: A Cognitive Observability Framework for the Reasoning of LLM-Powered Agents. *arXiv preprint arXiv:2411.03455* (2025).
- [27] Aqua Security. 2023. Tracee: Runtime Security and Forensics using eBPF. <https://github.com/aquasecurity/tracee>
- [28] Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O’Sullivan, and Hoang D. Nguyen. 2025. Multi-Agent Collaboration Mechanisms: A Survey of LLMs. *arXiv preprint arXiv:2501.06322* (2025).
- [29] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. AutoGen: Enable Next-Gen Large Language Model Applications. <https://github.com/microsoft/autogen>
- [30] Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. 2024. InjecAgent: Benchmarking Indirect Prompt Injections in Tool-Integrated Large Language Model Agents. In *Findings of the Association for Computational Linguistics (ACL Findings)*. doi:10.48550/arXiv.2403.02691 arXiv:2403.02691.
- [31] Boyang Zhang, Yicong Tan, Yun Shen, Ahmed Salem, Michael Backes, Savvas Zannettou, and Yang Zhang. 2024. Breaking Agents: Compromising Autonomous LLM Agents Through Malfunction Amplification. *arXiv preprint arXiv:2407.20859* (2024). Introduces an "infinite loop" attack that forces agents to repeat actions until max iteration..
- [32] Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, Xiaozheng Lai, Dan Williams, and Andi Quinn. 2025. Extending Applications Safely and Efficiently. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 557–574.