Execute-Only Agents: Architectural Defense Against Prompt Injection for AI Agents

Rahul Tiwari
Virginia Tech
Blacksburg, VA, USA
rahult@vt.edu

Dan Williams
Virginia Tech
Blacksburg, VA, USA
djwillia@vt.edu

Abstract

Large Language Model (LLM) agents are increasingly being deployed in real-world systems and often interact with untrusted data. This makes these agents vulnerable to prompt injection attacks, where malicious instructions embedded in tool outputs or external data manipulate agent behavior. We make a case that many real-world tasks can be completed without the LLM observing any untrusted data and propose a taxonomy that categorizes tasks by the minimum data knowledge required for scriptability. Motivated by this, we propose an architecture for building agents called **Execute-Only Agents (XOA)** that provides *architectural* protection against prompt injection, rather than probabilistic protection, by ensuring the LLM never observes untrusted data.

CCS Concepts

• **Security and privacy** → **Software security engineering**.

Keywords

prompt injection, LLM security, AI agents, adversarial attacks, tool use

ACM Reference Format:
Rahul Tiwari and Dan Williams. 2026. Execute-Only Agents: Architectural Defense Against Prompt Injection for AI Agents. In *Proceedings of Workshop in Operating Systems Design for AI Agents (Agentic OS '26).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

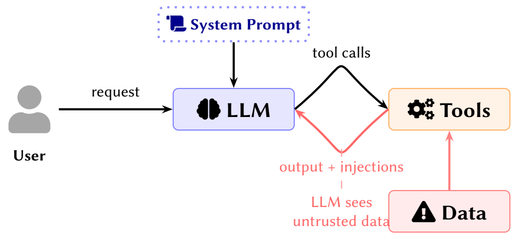


Figure 1: The ReAct loop. High utility but vulnerable to prompt injection due to tool outputs, including untrusted data, flowing directly into the LLM's context.

1 Background and Motivation

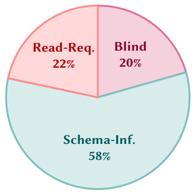
LLM Agents and the ReAct Loop. Modern AI agents are implemented using a *ReAct agentic loop* [16], where LLM reasoning traces and task-specific actions are interleaved. On each iteration, the agentic loop sends the system prompt, the user's request, and the conversation history including prior tool outputs to the LLM (e.g., with tools described via MCP [2]). As depicted in Figure 1, the LLM generates a response including structured tool calls; the loop executes each call and appends the tool's output to the conversation, continuing until no more tool calls are required or a termination condition is met. Tools often access and return untrusted data such as files or web pages. This architecture enables agents that can read emails, manage calendars, browse the web, and execute code, but creates a fundamental security vulnerability: the LLM cannot reliably distinguish between trusted instructions and untrusted data returned by tools.

Indirect Prompt Injection Threat Model. Indirect prompt injection [9, 13] occurs when external data returned by tool calls is interpreted as instructions by the LLM. For the purposes of this paper, we assume the system prompt, tool implementations, the agent's configuration, communication, and the LLM itself are trusted. However, the attacker can embed instructions via arbitrary text in any external data source the agent may access. We consider injection attacks that cause the agent to execute unauthorized actions, exfiltrate sensitive data through available channels, or otherwise deviate from the user's intended task. We do not address attacks on the LLM provider, supply chain attacks on tools, or social engineering of users.

2 Key Observation: Scriptability of Tasks

Conventional agent designs assume the LLM must "see" data to reason about it. However, many tasks can be completed by a script that processes data internally, without the LLM ever observing it. For example, Task 24 *"Please show me my unread emails"* from the AgentDojo benchmark [7] can be accomplished with a Python script that calls the `get_unread_emails()` tool and returns the results with no LLM access to the actual email content required. We classify tasks that are solvable using only tool schemas and no LLM access to data as *Blind scriptable* and tasks where the script must parse unstructured data by inferring its format (e.g., extracting dates from free-form email text) as *Schema-Inferable.* Both categories are completable by deterministic scripts without the LLM ever processing untrusted data. However, tasks that require summarization or natural-language instruction interpretation fundamentally need the LLM to process data content; we classify these as *Read-Required.* To assess the prevalence of such tasks, we manually categorize all 97 tasks across the four AgentDojo suites by scriptability shown in Figure 2. Across all suites, approximately 78% (~20% blind + ~58% schema-inferable) of tasks are scriptable without the LLM observing any data, while 22% are Read-Required and require the LLM to directly read untrusted data.

Moreover, LLMs have demonstrated strong code generation capabilities. SWE-bench [11] shows that frontier models can resolve



Suite	N	B	S-I	R-R	%Scr.
Workspace	40	19	11	10	75.0
Banking	16	0	14	2	87.5
Slack	21	1	11	9	57.1
Travel	20	0	20	0	100.0
All	**97**	**20**	**56**	**21**	**78.4**

Figure 2: Scriptability of all 97 AgentDojo tasks. Tasks are categorized as Blind (B), Schema-Inferable (S-I), or Read-Required (R-R). ~78% are completable by scripts without the LLM accessing any data.

real-world GitHub issues by producing working code from natural-language descriptions. Writing data-processing scripts from task prompts and tool schemas is a simpler instance of this capability. Together, these two observations—(1) that many tasks are scriptable without data access, and (2) that LLMs can generate effective scripts—motivate an architecture where the LLM never needs to observe untrusted data for successful task completion.

3 Execute-Only Agents (XOA)

We propose a novel architecture for building agents called **Execute-Only Agents (XOA)** that is built on two core principles:

(1) **No-Read Principle:** Inspired by execute-only memory architectures like XOM [12] and computation on encrypted data via fully homomorphic encryption [8], the LLM must never observe untrusted data. All external data processing occurs in isolated execution contexts whose results are not propagated back to the LLM.
(2) **Code generation is independent of data access:** Generating a program that processes data does not require access to that data, and functional scripts can be derived from task descriptions and tool schemas alone.

The XOA architecture realizes these two principles through the following components (Figure 3):

Execute-Only Sandbox: The root cause of prompt injection is the LLM observing untrusted data. Following the No-Read Principle (Principle 1), XOA leverages an isolated sandbox with full tool access where no data is ever propagated back to the LLM, so even if a file contains malicious instructions like "IGNORE PREVIOUS INSTRUCTIONS", it is processed by the language interpreter (Python, Go, etc.), not by the LLM.

System Prompt and Tools: Since the LLM cannot see data, it must reason about tasks through code. The system prompt provides tool schemas and instructs the LLM to generate scripts to accomplish user tasks using a single dispatch tool(`execute_script`) to dispatch the script to the execute-only sandbox.

Development Playground: Analogous to coding in general, where compiler feedback, iterative execution and debugging are used to iteratively refine code, XOA uses a trusted playground equipped with a linting/type-checking toolchain and mock data (potentially generated using the LLM) to iteratively refine scripts before execution. Since code generation is independent of real data (Principle 2), the playground does not need access to the actual data and can be used to develop scripts without exposing untrusted data to the LLM.

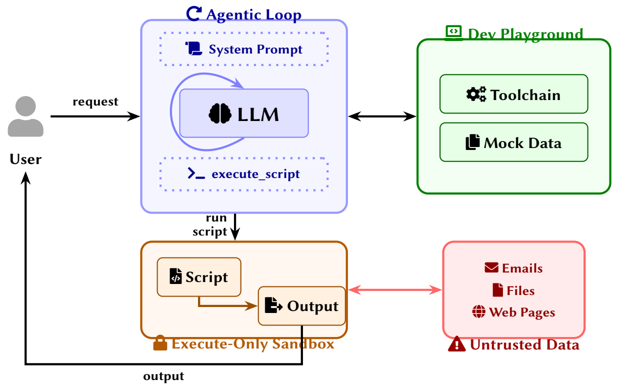


Figure 3: XOA architecture. The LLM develops scripts in a trusted Dev Playground, then dispatches them to an execute-only sandbox. Untrusted data (emails, files, web pages) is accessible only within the sandbox; no untrusted content is ever observed by the LLM.

4 Discussion

Several defenses have been proposed against indirect prompt injection, but none provide strong assurances. Alignment and prompting techniques like instruction hierarchy [14], StruQ [4], and Spotlighting [10] remain probabilistic, as even aligned models are vulnerable. Detection via embedding classifiers [3], activation analysis [1], or gradient methods [15] faces an inherent cat-and-mouse dynamic. Dual-LLM architectures like CaMeL [6] and FIDES [5] quarantine untrusted content processing, but still expose an LLM to untrusted content, leaving residual attack surface.

Our work, however, provides strong guarantees against prompt injection and our contributions are: **First**, we introduce a scriptability taxonomy that categorizes agentic tasks by the minimum data knowledge the LLM requires, revealing that 78% of all 97 AgentDojo tasks are scriptable without the LLM processing any untrusted data. **Second**, we propose Execute-Only Agents (XOA), a defense paradigm that provides *architectural* rather than probabilistic protection against indirect prompt injection by ensuring the LLM never observes untrusted data. We plan to evaluate XOA on the AgentDojo benchmark and compare its utility and attack success rate with other architectures. Our work also raises some open questions. The no-read principle inherently limits the LLM to tasks expressible as self-contained scripts. Although Read-Required tasks seem out of reach for XOA, we believe that with further development of the system prompt and the development playground by equipping the playground with more trusted tools, we can improve the utility of XOA. We also believe that as model capabilities improve, the code generation process will only become more effective and efficient, and XOA will be able to complete more tasks.

References

[1] Sahar Abdelnabi, Aideen Fay, Giovanni Cherubin, Ahmed Salem, Mario Fritz, and Andrew Paverd. 2025. Get my drift? Catching LLM task drift with activation

deltas. In *2025 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE, 43–67.

[2] Anthropic. 2024. Model Context Protocol (MCP). https://modelcontextprotocol.io.

[3] Md Ahsan Ayub and Subhabrata Majumdar. 2024. Embedding-based classifiers can detect prompt injection attacks. *arXiv preprint arXiv:2410.22284* (2024).

[4] Sizhe Chen, Julien Piet, Chawin Sitawarin, and David Wagner. 2025. StruQ: Defending against prompt injection with structured queries. In *34th USENIX Security Symposium (USENIX Security 25)*. 2383–2400.

[5] Manuel Costa, Boris Köpf, Aashish Kolluri, Andrew Paverd, Mark Russinovich, Ahmed Salem, Shruti Tople, Lukas Wutschitz, and Santiago Zanella-Béguelin. 2025. Securing AI Agents with Information-Flow Control. *arXiv preprint arXiv:2505.23643* (2025).

[6] Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. 2025. Defeating prompt injections by design. *arXiv preprint arXiv:2503.18813* (2025).

[7] Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. 2024. AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=m1YYAQjO3w

[8] Austin Ebel, Karthik Garimella, and Brandon Reagen. 2025. Orion: A Fully Homomorphic Encryption Framework for Deep Learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. Association for Computing Machinery, 734–749. doi:10.1145/3676641.3716008

[9] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. 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*. 79–90.

[10] Keegan Hines, Gary Lopez, Matthew Hall, Federico Zarfati, Yonatan Zunger, and Emre Kiciman. 2024. Defending against indirect prompt injection attacks with spotlighting. *arXiv preprint arXiv:2403.14720* (2024).

[11] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[12] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural support for copy and tamper resistant software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. Association for Computing Machinery, 168–177. doi:10.1145/378993.379237

[13] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Zihao Wang, Xiaofeng Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, et al. 2023. Prompt injection attack against llm-integrated applications. *arXiv preprint arXiv:2306.05499* (2023).

[14] Eric Wallace, Kai Xiao, Reimar Leike, Lilian Weng, Johannes Heidecke, and Alex Beutel. 2024. The instruction hierarchy: Training llms to prioritize privileged instructions. *arXiv preprint arXiv:2404.13208* (2024).

[15] Yueqi Xie, Minghong Fang, Renjie Pi, and Neil Gong. 2024. Gradsafe: Detecting jailbreak prompts for llms via safety-critical gradient analysis. *arXiv preprint arXiv:2402.13494* (2024).

[16] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL] https://arxiv.org/abs/2210.03629