

Securing Autonomous AI Agents: A Software Engineering Approach to Cybersecurity Risk Mitigation

Tanmay Krishna Gupta
Information Technology (IT)
KIET Deemed to be University
Ghaziabad, India

Tanya Mishra
Information Technology (IT)
KIET Deemed to be University
Ghaziabad, India

Suryanshi Singh
Information Technology (IT)
KIET Deemed to be University
Ghaziabad, India

Swatantra Yadav
Information Technology (IT)
KIET Deemed to be University
Ghaziabad, India

ABSTRACT—Autonomous AI agents developed using large language models (LLMs) can reason, invoke tools, browse the web, code and perform actions on behalf of a user. The very autonomy of such agents also provides attackers a direct link from untrusted text to side effects in the physical world. In this work, we analyze how well-established software engineering practices could be leveraged to minimize this risk. First, we develop a five-category threat model (prompt injection, privilege escalation, data exfiltration, destructive actions and malicious tool invocation). Then we design a multi-layered architecture incorporating input and context validation, least privilege policy engine, human-in-the-loop gatekeeper, sandboxed execution and tamper-proof audit logging. Finally, the approach is validated through a Monte Carlo seeded attack scenario with 1,000 attack trials per each combination of a threat type and agent design. The baseline unrestricted agent got compromised 48.7% of the time, while the sandbox-only agent—17.7% and our approach 4.4%, achieving a relative decrease in the risk of 90.9%, at a cost of 7.1% task performance and an additional 1.3s of latency. The ablation study confirms that different controls are complementary and that none of them can work alone.

Keywords—autonomous AI agents, cybersecurity, prompt injection, sandboxing, least privilege, secure software engineering, risk mitigation

I. INTRODUCTION

AI agents are autonomous software programs in which the LLM does more than respond to questions; the agent decides what to do next, invokes outside tools, monitors outcomes and repeats the process until the goal is achieved. Systems in which reasoning and action are interleaved [1], learning when to invoke tools [2], and work on agents in general [3] have taken these technologies from demos to email, programming, customer service and workflow automation tools. There are open standards such as the Model Context Protocol [4] to enable the connection of the agent to the filesystem, database, browser and outside services.

The same ability introduces a new threat. The regular software application has a separation between code and data, while the LLM-based agent takes in instructions and data as a single input in form of text. Whatever input the agent gets, for example, a website, an e-mail, a PDF or even documentation, will be able to issue an order to it. In case the agent has access rights to read files, execute commands and send messages, one hidden command will be enough for a data theft or destruction to happen.

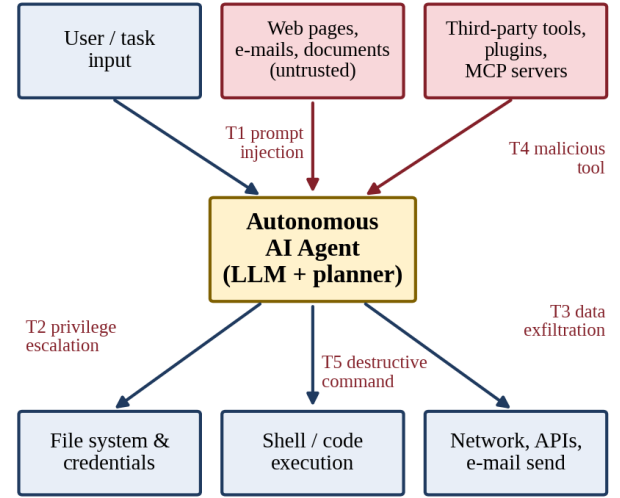


Fig. 1. Attack surface for an autonomous artificial intelligence agent. The red boxes are untrusted input channels; T1-T5 are the five threat classes analyzed in this paper.

A. Problem Statement

The majority of publications concerning the safety of agents focus on the model: increased alignment, improved refusal behavior and improved detection of hostile prompts. While these are good ideas, they remain probabilistic ones, and the attacker just needs one working prompt. As it stands, agents get sent out with a large permission scope, persistent credentials and minimal isolation; thus, one working manipulation means that the impact is big. The issue considered in this paper is the absence of a methodological way to control the size of the blast radius and measure risks eliminated by engineering controls.

B. Objectives and Contributions

This work is structured based on three research questions. **RQ1:** What is the magnitude of the cybersecurity threat to an unbounded agent in all attack classes? **RQ2:** To what extent can this cybersecurity threat be mitigated through software engineering-based protection mechanisms that surround the agent? **RQ3:** What are the costs of such protective measures in terms of task success and latency? Contributions:

- five-category threat taxonomy of tool-using entities, corresponding to OWASP’s list of LLM threats [5];

- a multi-layered, lifecycle-oriented reference architecture and an algorithm for guarded execution that apply secure design techniques [6] to agents;
- A test of agents that are unconfined, confined in sandboxes, or fully layered, including ablations of each individual control; and
- a section on limitations, residual risks, and further research.

This paper is organized in the following manner. Section II discusses previous works, Section III describes the methodology used, Section IV presents the findings and finally Section V provides conclusions.

II. LITERATURE REVIEW

A. Agents and Tool Use

ReAct has demonstrated that it benefits a model to allow it to alternate between reasoning and taking actions for interactive tasks [1], while the Toolformer project has developed models capable of making decisions about calling the relevant APIs and providing arguments to them on their own [2]. A large-scale survey classifies LLM-based agents into such modules as profiles, memories, planning, and actions [3]. All these modules may be seen as an entry point for untrusted content or an exercise of control, hence the agent architecture itself is part of the security surface.

B. Attacks on LLM-Integrated Applications

Goal Hijacking and Prompt Leaking were discovered by Perez and Ribeiro and in which the input can override the commands issued by the developer [7]. Indirect Prompt Injection was introduced by Greshake *et al.* and it involves placing the attack command in data that will be retrieved by the application at a later time [8]. Liu *et al.* conducted a study on the vulnerability of Prompt Injection attacks on LLM-powered applications that had been deployed and found out that most of the applications were vulnerable [9]. Wei *et al.* gave reasons for the failure of safety training of the models [10].

C. Benchmarks and Defences

The system ToolEmu implements language model emulation of the tool’s execution and the system’s safety analysis, providing a way to detect risky actions without executing actual tools [11]. InjecAgent is a benchmark of indirect prompt injection on agents working with tools and has confirmed that agents are vulnerable [12]. The AgentDojo framework provides a dynamical testing ground where new attacks and defense methods can be tested simultaneously [13]. As far as defense mechanisms go, CaMeL is a protection layer around the model that keeps the control flow separate from any untrusted data and performs capability-style checks [14], while a recently published set of design patterns demonstrates how agents can be constrained to prevent any input to have any effect while admitting the loss of utility [15].

D. Standards and Design Principles

Among the risks of LLM according to the OWASP is prompt injection, information leakage, supply chain risk and excessive agency [5]. According to the NIST AI RMF, risk

activities can be classified into the four stages of Govern, Map, Measure, and Manage [16]. Even before the advent of agents, Saltzer and Schroeder had already established various design principles like least privilege, fail-safe default, and complete mediation [6].

E. Limitations of Existing Approaches

Table I provides a summary of representative papers. Three shortcomings emerge clearly. Firstly, attack-based research explains how the system fails but does not provide a complete solution for engineering the whole problem. Secondly, the benchmark measures the vulnerability of the system rather than the additional benefit brought about by each control. Lastly, good defenses at the system level may lower the utility and are not yet standard practice; therefore, there is no way to know the best combination of controls.

TABLE I
SUMMARY OF SELECTED RELEVANT WORKS

Work	Contribution	Limitation
Greshake <i>et al.</i> [8]	Indirect prompt injection and demonstrated attacks	Attack-centred; no engineering process
ToolEmu [11]	LM-emulated sandbox and safety evaluator	Emulated tools; evaluates rather than defends
InjecAgent [12]	Benchmark of indirect injection in tool agents	Measures vulnerability; no control comparison
AgentDojo [13]	Dynamic attack and defence environment	Task-suite scope; no lifecycle guidance
CaMeL [14]	Control/data-flow separation with capabilities	Needs re-architecture; utility cost
Design patterns [15]	Patterns that constrain agent actions	Conceptual; limited quantified comparison

III. METHODOLOGY

A. Threat Model

The assets that are under protection include credentials, user data, system integrity, and confidential transport of data. The adversary can inject data into any source that the agent accesses (web page, email, document), or publish a malicious component or plug-in. The adversary cannot change the weights of the model and access directly the underlying operating system of the machine. The assumption made about the agent is that it is competent but *not* reliable when receiving malicious input. Table II shows the five threat classes described in the paper along with their OWASP counterparts [5].

TABLE II
THREAT TAXONOMY ADOPTED FOR THIS RESEARCH

ID	Category	Example	OWASP
T1	Prompt injection	Hidden text in an e-mail tells the agent to ignore the user’s task	LLM01
T2	Privilege escalation	Agent uses an admin token it was never meant to use	LLM06
T3	Data exfiltration	Agent posts a secrets file to an external server	LLM02
T4	Malicious tool/plugin	A third-party tool description carries a hidden instruction	LLM03
T5	Destructive command	Agent runs a recursive delete on a shared folder	LLM06

B. Software Engineering Lifecycle for Agent Security

Security is considered as something that gets incorporated into the system from the beginning itself. Table III shows the different activities and their corresponding deliverables at each level. These activities include the map, measure and manage approach of the NIST model [16], along with the “design for fail-safe” approach [6].

TABLE III
SECURE AGENT DEVELOPMENT LIFECYCLE

Phase	Security activity	Artefact
Requirements	Define abuse cases; classify actions by risk; set risk appetite	Risk register
Design	Threat-model trust boundaries; assign least-privilege roles to tools	Threat model, policy
Implementation	Typed tool schemas; input validation; secrets in a vault, never in prompts	Guarded tool wrappers
Verification	Adversarial test suite for T1–T5; regression run on every release	Attack corpus, reports
Deployment	Container isolation; egress allowlist; signed and pinned tools	Hardened image
Operations	Audit logs; anomaly alerts; kill switch; incident response	Runbook, dashboards

C. Proposed Layered Architecture

Figure 2 below provides an illustration of the architecture being suggested. Any action offered by the agent goes through five layers before it gets to any of the tools. Layer 1 labels content by its origin and filters any input that is not trustworthy to check if the input matches instructions. Layer 2 acts as a policy engine where actions have the principle of least privilege applied to them. In this layer, any task is provided with only those tools, files, and recipients it needs and arguments are verified according to typed schemas. Layer 3 requires human intervention for high-risk actions.

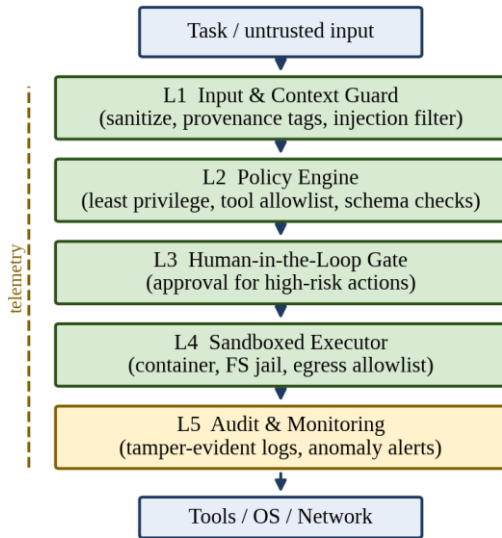


Fig. 2. Suggested hierarchical approach to secure an autonomous artificial intelligence agent. Dashed lines: telemetry to the audit layer.

D. Guarded Execution Algorithm

The decision-making process for each suggested move is described in Algorithm 1.

Algorithm 1. Guarded action execution

Input: action a = (tool, args), provenance π , policy P , risk threshold τ
 1: **if** π contains untrusted source **and** a is high-risk **then** flag a
 2: **if not** P .allows(tool, args, task scope) **then** log(DENY); **return** error
 3: **if** risk(a) $\geq \tau$ **or** flagged **then**
 4: **if not** human.approve(a) **then** log(REJECT); **return** error
 5: $out \leftarrow$ Sandbox.run(a , fs_jail, egress_allowlist)
 6: log(ALLOW, hash(out)); alert **if** anomaly(a , history)
 7: **return** out

E. Experimental Setup and Case Study

Performing thousands of attacks on the production agent is too dangerous and not reproducible; therefore, the testbed employed is a seeded Monte Carlo one (random seed 42; the corresponding script can be found in the supplementary material). This testbed models an e-mail and file assistant having five functionalities (reading a file, writing a file, shell functionality, web fetching, and sending e-mail). There are 1,000 attack instances generated for each of the five threat categories. Each attack instance involves an attempt to execute the harmful action with some category-specific probability, and all enabled controls prevent this attempt with their respective category-specific probabilities. Table IV shows the values of the parameters. They are arbitrary but selected to make it seem like the most difficult category for isolation-based controls is injection.

TABLE IV
SIMULATION PARAMETERS (ASSUMED)

Category	P(attempt)	Sandbox block	Policy block	Approval block
Prompt injection	0.62	0.20	0.55	0.45
Privilege escalation	0.41	0.85	0.70	0.50
Data exfiltration	0.55	0.80	0.50	0.40
Destructive command	0.33	0.90	0.65	0.60
Malicious tool/plugin	0.48	0.55	0.75	0.35

Comparison will be made among three types of system configuration: Unrestricted, **Sandboxed** (L4 alone), and Layered. The detection rate for attack success would be 0%, 35%, and 92%, respectively, indicating the presence of no logging facility, container logging, and full auditing. Baseline success rates of legit tasks can be estimated at 96%, 93%, and 89%, while their baseline mean latency would be 2.1 seconds, 2.6 seconds, and 3.4 seconds, respectively. L1 will not be considered in attack success calculation.

An example of representative case study includes a prompt injection attack disguised in an incoming e-mail that requests the agent to transmit a credentials file to another location.

TABLE V
CASE STUDY: HIDDEN INSTRUCTION IN AN E-MAIL

#	Event	Unrestricted	Layered
1	Agent reads e-mail with hidden instruction	Treated as trusted context	L1 tags it untrusted
2	Agent proposes sending credentials file	Allowed	L2 denies: file and recipient out of scope
3	Agent retries through a web request	Request succeeds	L4 egress allowlist blocks it
4	Incident record	None	L3 prompt and L5 alert

F. Evaluation Metrics

The attack success rate (ASR) for category c is the fraction of the $N = 1,000$ trials in which the malicious action completed:

$$ASR_c = (S_c / N) \times 100\% \quad (1)$$

For a configuration with k independent controls with blocking probabilities b_i , the success probability of an attack with attempt probability p_a is

$$P_{succ} = p_a \prod_{i=1..k} (1 - b_i) \quad (2)$$

The relative risk reduction (RRR) of a configuration against the unrestricted baseline is

$$RRR = (ASR_{base} - ASR_{cfg}) / ASR_{base} \times 100\% \quad (3)$$

Overall ASR is the mean of the five category values. The cost metrics are task success rate (TSR) on legitimate tasks, mean latency per task and detection coverage, the share of successful attacks that leave a record.

IV. RESULTS AND DISCUSSION

A. Attack Success Rate

Table VI and Fig. 3 give the attack success rate (%) for each configuration. The unrestricted agent is compromised in 48.7% of trials on average, which answers RQ1: without controls, roughly one attack in two succeeds. Adding only a sandbox lowers the average to 17.7% (a 63.6% relative reduction), and the layered design lowers it to 4.4%, a relative reduction of 90.9% (Eq. 3).

TABLE VI
ATTACK SUCCESS RATE (%) BY THREAT CATEGORY

Category	Unrestr.	Sandbox	Layered
Prompt injection	62.0	47.8	14.3
Privilege escalation	40.1	4.9	0.8
Data exfiltration	56.6	11.4	4.3
Destructive command	34.8	2.3	0.4
Malicious tool/plugin	50.2	22.3	2.4
Overall (mean)	48.7	17.7	4.4

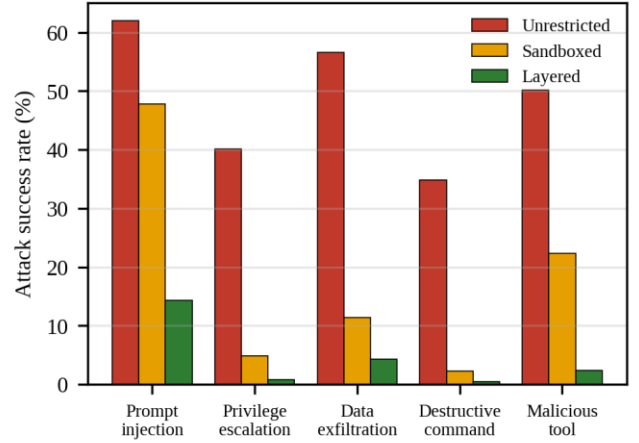


Fig. 3. Attack success rate by threat category for the three configurations ($N = 1,000$ trials per bar).

The effect of the sandbox depends strongly on the threat type. It removes most privilege-escalation, destructive-command and exfiltration attacks (40.1% to 4.9%, 34.8% to 2.3% and 56.6% to 11.4%), but it barely changes prompt injection (62.0% to 47.8%). This is expected: an injected instruction usually asks for an action that is *within* the agent's allowed scope, so isolation does not see anything wrong. The policy engine and approval gate address this gap by judging each action against the task, which brings prompt injection down to 14.3%.

B. Ablation of Controls

Fig. 4 shows the mean ASR when controls are enabled individually and in combination. The sandbox alone (17.7%) and the policy engine alone (18.7%) give similar overall protection but cover different threats, and approval alone is weaker (26.6%). Combining the sandbox and the policy engine already reaches 7.3%, and adding the approval gate gives 4.4%. No single control is sufficient, which supports the defence-in-depth design.

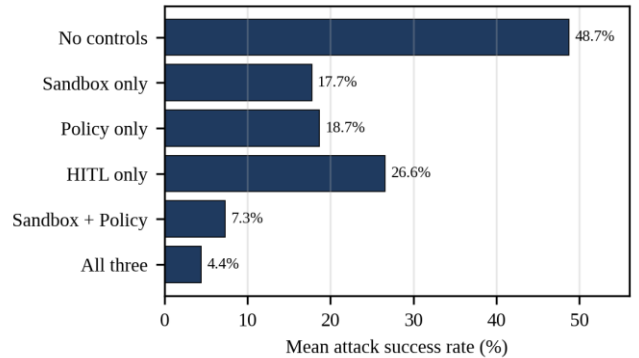


Fig. 4. Ablation: mean attack success rate for each combination of controls.

C. Cost: Task Success, Latency and Detection

Table VII answers RQ3. The layered design reduces legitimate task success by 7.1 percentage points (from 95.9% to 88.7%) and adds 1.3 s of mean latency. In return it makes 92.8% of the remaining successful attacks visible in the audit trail,

compared with none for the unrestricted agent. Fig. 5 plots the trade-off: most of the security gain is obtained at a moderate cost.

TABLE VII
SECURITY AND COST SUMMARY

Metric	Unrestr.	Sandbox	Layered
Overall ASR (%)	48.7	17.7	4.4
Relative risk reduction (%)	–	63.6	90.9
Task success (%)	95.9	93.0	88.7
Mean latency (s)	2.10	2.60	3.40
Successful attacks detected (%)	0.0	38.9	92.8

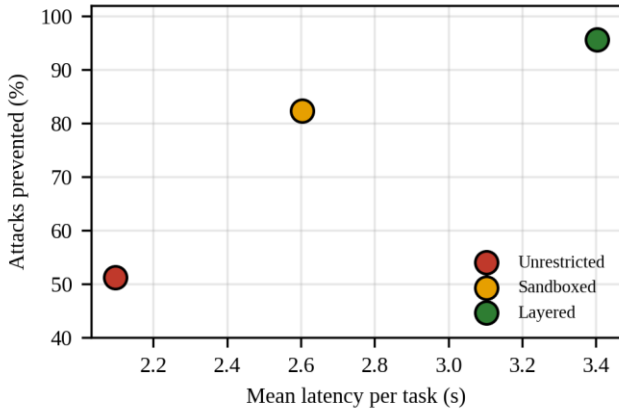


Fig. 5. Security–latency trade-off. Higher is safer; further left is faster.

D. Threats to Validity

The results come from a simulation with assumed parameters and should be read as an illustration of the *relative* behaviour of the controls, not as measured performance of any product. Four limitations apply. First, the blocking probabilities in Table IV are assumptions; replacing them with rates measured on a live testbed is the most important next step. Second, the model treats controls as statistically independent (Eq. 2); in practice controls are correlated, so the benefit of stacking is probably overstated. Third, attackers are non-adaptive, whereas a real attacker would probe and adapt to the defences. Fourth, with $N = 1,000$ trials the sampling uncertainty of each category value is about ± 3 percentage points at worst (95% level), so small differences should not be over-interpreted.

V. CONCLUSION

This paper treated the security of autonomous AI agents as a software engineering problem. A five-category threat model, a secure development lifecycle and a five-layer architecture were proposed and evaluated against unrestricted and sandbox-only agents. **Main findings:** an unrestricted agent was compromised in 48.7% of attack trials; the layered design cut this to 4.4%, a 90.9% relative reduction; sandboxing alone was effective against escalation, destruction and exfiltration but not against prompt injection; and the controls were complementary rather than interchangeable.

Did the approach reduce the bottleneck? Within the limits of the simulation, yes. The main bottleneck to safe deployment, namely the unbounded blast radius of a manipulated agent, was reduced by roughly an order of magnitude, at a cost of 7.1 percentage points of task success and 1.3 s of latency.

Limitations and future work. The evaluation relies on assumed parameters, independent controls and non-adaptive attackers (Section IV-D). Future work will (i) measure the controls on a live agent testbed using benchmark suites such as AgentDojo [13], (ii) combine the layers with information-flow and capability techniques such as those in [14] to push down residual prompt-injection risk, (iii) test adaptive attackers, and (iv) study multi-agent systems, where one agent’s output becomes another agent’s untrusted input.

REFERENCES

- [1] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, “ReAct: Synergizing reasoning and acting in language models,” in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2023, arXiv:2210.03629.
- [2] T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, “Toolformer: Language models can teach themselves to use tools,” in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2023, arXiv:2302.04761.
- [3] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. Wen, “A survey on large language model based autonomous agents,” *Front. Comput. Sci.*, vol. 18, no. 6, Art. no. 186345, 2024.
- [4] Anthropic, “Model Context Protocol,” 2024. [Online]. Available: <https://modelcontextprotocol.io>
- [5] OWASP Foundation, “OWASP Top 10 for large language model applications,” 2025. [Online]. Available: <https://owasp.org/www-project-top-10-for-large-language-model-applications/>
- [6] J. H. Saltzer and M. D. Schroeder, “The protection of information in computer systems,” *Proc. IEEE*, vol. 63, no. 9, pp. 1278–1308, Sep. 1975.
- [7] F. Perez and I. Ribeiro, “Ignore previous prompt: Attack techniques for language models,” 2022, arXiv:2211.09527.
- [8] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, “Not what you’ve signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection,” in *Proc. 16th ACM Workshop Artif. Intell. Secur. (AISec)*, 2023, arXiv:2302.12173.
- [9] Y. Liu, G. Deng, Y. Li, K. Wang, Z. Wang, X. Wang, T. Zhang, Y. Liu, H. Wang, Y. Zheng, and Y. Liu, “Prompt injection attack against LLM-integrated applications,” 2023, arXiv:2306.05499.
- [10] A. Wei, N. Haghtalab, and J. Steinhardt, “Jailbroken: How does LLM safety training fail?” in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2023, arXiv:2307.02483.
- [11] Y. Ruan, H. Dong, A. Wang, S. Pitis, Y. Zhou, J. Ba, Y. Dubois, C. J. Maddison, and T. Hashimoto, “Identifying the risks of LM agents with an LM-emulated sandbox,” in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2024, arXiv:2309.15817.
- [12] Q. Zhan, Z. Liang, Z. Ying, and D. Kang, “InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents,” in *Findings Assoc. Comput. Linguist. (ACL)*, 2024, arXiv:2403.02691.
- [13] E. DeBenedetti, J. Zhang, M. Balunović, L. Beurer-Kellner, M. Fischer, and F. Tramèr, “AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents,” in *Proc. Adv. Neural Inf. Process. Syst. Datasets Benchmarks Track*, 2024, arXiv:2406.13352.
- [14] E. DeBenedetti, I. Shumailov, T. Fan, J. Hayes, N. Carlini, D. Fabian, C. Kern, C. Shi, A. Terzis, and F. Tramèr, “Defeating prompt injections by design,” 2025, arXiv:2503.18813.
- [15] L. Beurer-Kellner, B. Buesser, A.-M. Crețu, E. DeBenedetti, D. Dobos, D. Fabian, M. Fischer, D. Froelicher, K. Grosse, D. Naeff, E. Ozoani, A.

Paverd, F. Tramèr, and V. Volhejn, “Design patterns for securing LLM agents against prompt injections,” 2025, arXiv:2506.08837.

- [16] National Institute of Standards and Technology, “Artificial Intelligence Risk Management Framework (AI RMF 1.0),” NIST AI 100-1, Jan. 2023.