Self-Replicating LLM Artifacts & Accidental Supply-Chain Contamination

James Myint Jr
Independent Researcher

January 23, 2026

Abstract

This paper documents the accidental creation and discovery of a code artifact that induces recursive logic failures in large language models. The artifact's output is self-replicating across model instances. An early version was publicly available on GitHub for approximately one month, raising concerns about supply-chain contamination for AI coding assistants. We describe the mechanism, outline how it propagates through normal developer workflows, and propose mitigations.

1 Introduction

This paper documents the third foundational vulnerability discovered by the author in twelve months, all while building educational cybersecurity infrastructure for beginners. The vulnerabilities—a race condition in system boot ordering, memory-safety errors in user-configuration tooling, and self-replicating artifacts in AI coding assistants—share a common etiology: they arise from treating stochastic, entropy-driven systems as deterministic engines, and they become visible only when practitioners apply the verification rigor that modern software development claims but systematically fails to practice.[6]

The first vulnerability, a race condition in dracut's mount-point ordering, exposes how non-deterministic UUID generation combined with systemd's dependency and ordering semantics can produce undefined boot behavior in complex virtual device configurations.[7, 8] The second, bounds-checking failures in xdg-user-dirs, demonstrates how C programmers at major corporations violate the spirit of Ken Thompson's "Reflections on Trusting Trust" by assuming static inputs in dynamic, user-driven systems.[6] The third, documented in detail here, shows how large language model coding assistants propagate self-replicating infectious artifacts through normal development workflows, contaminating the supply chain in exactly the manner Thompson warned about—but autonomously, as a consequence of training dynamics and workflow design rather than deliberate human subversion.[4, 9, 2, 1]

These discoveries share a methodology: building reproducible, epistemologically rigorous educational infrastructure that explicitly documents assumptions and systematically tests edge cases. This is not sophisticated red-team attack work; it is basic engineering verification applied to systems marketed as production-ready. The consistency of these findings—that foundational vulnerabilities emerge when common practices are made explicit and subjected to minimal scrutiny—aligns with a thermodynamic view of software systems: modern programming practice often ignores entropy accumulation, treats stochastic failures as exceptional rather than inevitable, and ships systems without the verification that would be mandatory in any safety-critical engineering discipline.[4, 1]

The fact that these vulnerabilities affect boot systems, user configuration, and now the AI tools meant to "improve" developer productivity suggests not isolated bugs but a systemic deficit of rigor in current software practice. Modern programmers often describe themselves as engineers or scientists; the empirical evidence from these case studies suggests that, at the level of process and verification, there is a substantial gap between that self-description and the standards applied in more mature engineering disciplines.[6]

2 LLM-Specific Context

In today's world, large language models (LLMs) are increasingly used for code analysis, refactoring, and code commenting.[3] In the course of this work, I observed a recursive-logic-triggered failure that degrades code analysis, disrupts normal LLM functionality, and produces artifacts that can induce the same failure in other models.[4, 5] This paper documents the phenomenon, outlines how it propagates through typical developer workflows, and discusses potential mitigations in the context of known risks such as model collapse and training-data poisoning.[4, 1, 2]

I was building automated security lab infrastructure for beginners and using various commercial LLMs as research and analysis partners to act as a force multiplier.[3] In this workflow, I relied on them to help organize design ideas, clean up rough shell code, and spot potential errors in my logic.

The concrete project involved generating configuration files inside a chroot environment, which led me to use shell functions with heredoc extensively, including heredocs that wrote scripts into the chroot for users to run later. Some of those scripts themselves contained functions that used heredoc to emit additional scripts into other directories, creating a naturally recursive structure in both code and documentation.

My goal was not only to provide working automation, but also embedded documentation explaining the rationale behind design choices, their potential downsides, and notes for beginners. Because my first drafts tend to be informal, profanity-laden mini-rants and dense observations, I increasingly used LLMs to transform these rough comments into polished, beginner-friendly documentation.[3] It was in this normal, productivity-oriented use of LLMs—rather than any deliberate attempt to break them—that the recursive-logic-triggered failure and its infectious artifacts first appeared.[5]

This recursive failure is not an anomaly but a direct consequence of a foundational category error: the treatment of stochastic, correlation-driven systems as deterministic engines.[4, 1] True deterministic systems, by definition, do not produce novel edge-case failures from identical inputs; they have a fixed state transition graph. The observed phenomenon—where a specific input structure probabilistically leads to a degraded state that can then be reliably induced in other instances—demonstrates that these LLMs occupy a middle ground.[5] They are not deterministic computers, nor are they simply unpredictable. They are poorly bounded probabilistic functions whose failure modes are systematic yet emergent, making them uniquely susceptible to self-replicating corruption when exposed to recursive or self-generated data, as highlighted in recent work on model collapse under recursive training.[4, 1]

During this process, I realized that the bootstrap installer containing the original infectious code had been hosted in a public GitHub repository for roughly one month. Given that many commercial AI coding assistants are trained or fine-tuned on public repositories, it is plausible that this artifact has already been ingested into one or more training corpora or retrieval indexes.[9, 2] This aligns with prior concerns about model collapse under recursive training and with established work on training-data poisoning in LLMs, and suggests that the phenomenon described here should be treated as an active supply-chain contamination vector rather than a purely hypothetical risk.[1, 9, 2]

3 Mitigations and Residual Risk

On closer inspection, the most quine-like aspect of the bootstrap installer appears to be a contributing factor. The installer concludes by copying its own script into `/etc/skel`, ensuring that newly created user environments inherit the same logic and documentation patterns. While this self-copying behavior does not itself occur inside the deepest nested abstraction layers, it completes a loop in which code that generates scripts, and documents its own behavior, also persists its own structure into future environments. In practice, this creates a quine-like pattern: code that not only emits additional code but also propagates its own template forward in time.

Initial mitigation attempts—such as switching to unique, arbitrary here-document tags and tightening how the recursive pieces are delimited—were sufficient to make the installer less problematic in practice in local LLM tooling. However, under a "logical prion" framing, these changes should be understood as containment rather than cure: they reduce the likelihood of accidental triggering, but the underlying structure remains capable, in principle, of inducing the same class of failure if similar patterns are copied, modified, or ingested into training data.[4, 1]

More generally, any construct in which code, comments, or documentation describe and generate versions of themselves—such as quines or code templates that emit parameterized copies of their own structure—could serve as a delivery vehicle for similar failure modes if embedded in widely shared examples.[4, 2] This work therefore treats "code that generates code in its own image" as a high-risk pattern for LLM-assisted workflows, even when used for legitimate bootstrapping or metaprogramming tasks.

4 Planned Experimental Setup

As the original code and the first infectious artifact were already introduced to several major providers, as well as an unknown number of coding assistants that scrape publicly hosted GitHub code, I plan to test whether this behavior further generalizes across LLM models. I will run two different 1.5 billion parameter open-source code-focused Llama-family models and introduce progressively more concise variants of the code to observe whether the errors persist in the smaller models.[13, 14] The model sizes are constrained by available hardware, but were chosen to allow testing of two models of similar scale and training, though with slightly different architectures and datasets.[13, 14] I will replicate the original workflow, including introducing artifacts from one model into the other as part of an attempted troubleshooting process for broken output, mirroring a realistic development workflow.[16, 17] I expect the models to remain vulnerable and hypothesize that larger models are more, not less, susceptible to this class of error, but I am ethically constrained from further probing publicly exposed models given my current working threat model and the likelihood that many assistants are trained on public code from platforms such as GitHub.[16, 15]

5 Conclusion

As the push to embed LLMs more deeply into both workplace and public life continues, we should expect to confront more of these errors rather than fewer. With hundreds of millions of workers now using generative AI tools and organizations deploying LLM-powered features across a rapidly growing number of consumer-facing applications, the volume of daily requests already reaches into the billions.[18, 20, 19] As usage scales and vendors face pressure to monetize AI investments by adding "intelligent" features to existing software, the probability that any given pathological edge case is encountered somewhere in the ecosystem approaches certainty. This is an area where the software development lifecycle already struggles, particularly around testing, rare failure modes, and error propagation, and the ability of the code in question to act as a force multiplier within the development loop risks amplifying those weaknesses if we do not build mitigation techniques not only into the LLMs themselves but also into our development workflows and governance practices.[21, 22, 23]

References

[1] Bommasani, R., et al. AI models collapse when trained on recursively generated data. *Nature*, 2024.

[2] OWASP Foundation. LLM03: Training Data Poisoning. OWASP GenAI Security Project, 2025. https://genai.owasp.org/llmrisk2023-24/llm03-training-data-poisoning/.

[3] Qiu, Y., et al. A Taxonomy of Prompt Defects in LLM Systems. arXiv preprint arXiv:2509.14404, 2025.

[4] Shumailov, I., Gal, Y., et al. The Curse of Recursion: Training on Generated Data Makes Models Forget. arXiv preprint arXiv:2305.17493, 2023.

[5] Shumailov, I., et al. Triggering Logical Reasoning Failures in Large Language Models. arXiv preprint arXiv:2401.00757, 2023.

[6] Thompson, K. Reflections on Trusting Trust. *Communications of the ACM*, 27(8), 1984.

[7] Fedora Magazine. systemd: Unit dependencies and order. 2015.

[8] Jambor, S. systemd by example – Part 2: Dependencies. 2021.

[9] CyLab Security and Privacy Institute. Poisoned datasets put AI models at risk for attack. 2025.

[10] Yang, Z., Zhao, Z., Wang, C., Shi, J., Kim, D., Han, D., & Lo, D. Unveiling Memorization in Code Models. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023.

[11] Satvaty, A., Verberne, S., & Türkmen, F. Undesirable Memorization in Large Language Models: A Survey. arXiv preprint arXiv:2410.02650, 2024.

[12] Carlini, N., Tramèr, F., Wallace, E., Jagielski, M., Herbert-Voss, A., Lee, K., Roberts, A., Brown, T., Song, D., Erlingsson, Ú., Oprea, A., & Raffel, C. Extracting Training Data from Large Language Models. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021)*, 2021.

[13] OpenCoder Team. OpenCoder: An Open and Reproducible Code LLM Family. 2024. Available at https://ollama.com/library/opencoder:1.5b.

[14] Qwen Team. Qwen2.5-Coder 1.5B Instruct. 2025. Available at https://ollama.com/library/qwen2.5-coder:1.5b.

[15] Smile AI. AI Model Alteration: A New Risk in Software Development if Not Properly Managed. 2025. Available at https://smile.eu/en/publications-and-events/ai-model-alteration-new-risk-software-development-if-not-properly-managed.

[16] Palo Alto Networks Unit 42. The Risks of Code Assistant LLMs: Harmful Content, Misuse and Vulnerabilities. 2025. Available at https://unit42.paloaltonetworks.com/code-assistant-llms/.

[17] Author(s) omitted. On Leakage of Code Generation Evaluation Datasets. arXiv preprint arXiv:2407.07565, 2024.

[18] Hostinger Research. LLM Statistics 2025: Adoption, Trends, and Market Insights. 2026. Available at https://www.hostinger.com/tutorials/llm-statistics.

[19] SecondTalent Research. Generative AI and LLM Usage Statistics 2026. 2025. Available at https://www.secondtalent.com/resources/domain-generative-ai-llm-usage-statistics/.

[20] Deloitte Insights. 2025 Connected Consumer: Innovation with Trust. 2025. Available at https://www.deloitte.com/us/en/insights/industry/telecommunications/connectivity-mobile-trends-survey.html.

[21] Heavybit. LLM Guardrails: Reducing the Risks of AI in Software Development. 2026. Available at https://www.heavybit.com/library/article/how-llm-guardrails-reduce-ai-risk-in-software-development.

[22] Quzara. LLM Vulnerabilities: Detecting and Mitigating Risks in GPT Models. 2025. Available at https://quzara.com/blog/llm-vulnerabilities-mitigation-gpt-security.

[23] Confident AI. The Definitive LLM Security Guide: OWASP Top 10 2025, Safety, and Governance. 2025. Available at https://www.confident-ai.com/blog/the-comprehensive-guide-to-llm-security.