Preprints.org

Article

Not peer-reviewed version

Dependency Risk Auditing for LLM-Generated Projects via Software Supply Chain Analysis

Seungmin Lee , Hyejin Choi , Jinwoo Park [*]

Posted Date: 2 February 2026

doi: 10.20944/preprints202602.0007.v1

Keywords: software supply chain; dependency auditing; CVE analysis; LLM-generated projects; secure package selection

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2026 doi:10.20944/preprints202602.0007.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Dependency Risk Auditing for LLM-Generated Projects via Software Supply Chain Analysis

Seungmin Lee, Hyejin Choi and Jinwoo Park *

School of Computer Science and Engineering, Seoul National University, Seoul 08826, South Korea
* Correspondence: j.park@snu.ac.kr

Abstract

LLM-generated code often pulls external dependencies without evaluating their security posture. We present an automated supply-chain auditing system that inspects dependency versions, known CVEs, integrity guarantees, and transitive dependency trees. Applied to 500 LLM-generated mini-projects, the audit system identifies risky dependencies in 72% of cases and provides secure alternatives. Integrating the system into LLM workflows decreases usage of vulnerable packages by 67%. This demonstrates that pairing LLM code generation with supply-chain intelligence is critical for preventing inadvertent security issues.

Keywords: software supply chain; dependency auditing; CVE analysis; LLM-generated projects; secure package selection

---

1. Introduction

Large language models (LLMs) have become widely adopted for generating code across a broad range of software development tasks, including web applications, system utilities, and lightweight software components. While these models significantly lower the barrier to development and improve productivity, recent empirical studies consistently report that LLM-generated code often exhibits security weaknesses, even when models are prompted with explicit safety instructions. Measurements across multiple model families show that more than half of generated programs contain missing validation checks, insecure defaults, or incomplete error handling mechanisms [1]. Additional analyses indicate that LLMs frequently reuse outdated programming patterns or overlook modern security guidelines, reflecting limitations in both training data and generation behavior [2]. Beyond general coding errors, several studies have highlighted systematic security failures in LLM-generated applications, particularly in web-facing contexts. Recurring issues have been observed in authentication logic, session management, and input sanitization, all of which can directly expose applications to exploitation when deployed without further review [3]. More recent research further suggests that these weaknesses are not isolated mistakes but structural tendencies of current-generation models, raising concerns about their suitability for direct use in production pipelines without automated safeguards [4]. In response, researchers have begun exploring approaches that guide LLMs away from insecure patterns, such as steering generation away from deprecated or unsafe APIs and encouraging safer alternatives during code synthesis [5].

At the same time, software security risks have increasingly shifted toward the software supply chain. Modern development relies heavily on public package ecosystems, with registries such as npm and PyPI collectively serving hundreds of billions of downloads each year [6]. This scale has made package ecosystems an attractive target for attackers, who exploit automated dependency resolution and developer trust to introduce malicious or compromised libraries into projects. Documented attacks include typosquatting, dependency confusion, and the insertion of malicious code into otherwise legitimate packages [7,8]. Studies further show that long and complex dependency chains amplify these risks, as vulnerabilities in transitive dependencies may remain unnoticed without specialized analysis tools [9]. In response, security agencies and industry organizations increasingly

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2026
doi:10.20944/preprints202602.0007.v1

recommend dependency auditing practices, including software bills of materials (SBOMs) and routine vulnerability scanning, to improve transparency and risk management in software projects [10]. These supply chain concerns become more pronounced in the context of LLM-assisted development. Code generation systems frequently insert import statements, recommend external libraries, or suggest specific package versions automatically. Industrial reports and academic analyses indicate that LLMs may hallucinate package names or recommend non-existent libraries, creating opportunities for attackers to upload malicious packages that match these generated names [11]. This phenomenon, often described as "slopsquatting," demonstrates how model-generated outputs can unintentionally expand the attack surface of public registries [12]. Other studies show that even when package names are valid, LLMs may favor unmaintained or vulnerable dependencies, especially when users directly copy generated code into projects without further inspection [13]. Research on training data poisoning further suggests that exposure to insecure or outdated examples can bias models toward unsafe dependency choices [14]. Despite growing awareness of these risks, existing security tools address only part of the problem. Supply chain security solutions primarily target established projects with persistent dependency manifests, focusing on vulnerability databases, metadata analysis, or behavior-based detection of malicious packages [15]. SBOM-related research evaluates how effectively current tools capture complete dependency graphs and how missing information affects vulnerability tracking [16]. These approaches, however, assume that dependencies are selected deliberately by developers and remain stable over time. They do not account for short-lived, rapidly generated code artifacts produced during interactive LLM sessions, nor do they integrate directly with model-generated outputs. In parallel, research on secure LLM code generation has concentrated on identifying and repairing weaknesses in the generated code itself. Prior work documents missing checks, unsafe control flow, and structural flaws across multiple programming languages [17]. Repair-oriented approaches attempt to modify generated code to improve correctness or security, but they typically treat dependencies as fixed and do not analyze the security properties of external packages introduced by the model [18]. Recent surveys acknowledge the importance of supply chain security in LLM-driven development but stop short of proposing automated mechanisms that connect dependency auditing with code generation workflows [19]. As a result, a clear gap remains between traditional supply chain security tools and the practical needs of projects created or influenced by LLMs.

This study seeks to bridge that gap by introducing an automated supply chain auditing system specifically designed for LLM-generated code. The proposed system analyzes declared dependencies at generation time, checks package versions against known vulnerabilities, evaluates integrity and maintenance indicators, and inspects transitive dependency trees to identify hidden risks. When issues are detected, the system recommends safer alternatives or mitigations that can be applied before code is integrated into a project. To evaluate its effectiveness, the system is applied to 500 LLM-generated mini-projects, revealing that 72% include at least one risky dependency. When incorporated into the code generation workflow, the system reduces the use of vulnerable packages by 67%. These results demonstrate that LLM-assisted development can be meaningfully combined with automated supply chain analysis, offering a practical path toward reducing unintentional dependency-related security exposure in modern software development.

2. Materials and Methods

2.1. Sample Description and Study Scope

This study examines 500 small projects generated by large language models. All samples were created using the same prompt style to keep task scope consistent. The projects contain code and dependency lists for common tasks in web development, scripting, and basic system utilities. Only samples with clear package names and version information were included. Projects with missing installation details or incomplete metadata were removed. The final dataset covers packages from

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2026 doi:10.20944/preprints202602.0007.v1

npm, PyPI, and system-level libraries, which provides a broad view of dependency behavior in generated code.

2.2. Experimental Design and Control Groups

The experiment compares two workflows. In the baseline workflow, the model generates code and dependencies without any review step. In the audited workflow, each generated project is checked by the supply-chain auditing system before evaluation. The auditing system inspects package versions, known vulnerabilities, and the full dependency tree. Both workflows use the same prompts and model parameters so that differences can be linked to the auditing step alone. Each experiment was repeated three times to reduce variation from the model s randomness.

2.3. Measurement Method and Quality Control

Each project is evaluated in three stages. First, all declared dependencies are extracted. Second, each dependency is matched to vulnerability records from public databases. Third, two reviewers classify dependencies as "safe," "outdated," or "vulnerable." A dependency is marked "risky" if it appears in a current CVE list or if the version is older than the fixed release cited in official advisories. A third reviewer resolves disagreements. To ensure stable evaluation, projects with inconsistent or unresolvable package data are excluded from the analysis.

2.4. Data Processing and Calculation

Dependency data is cleaned to standardize package names and version formats. Aliases and duplicated entries are merged. The main indicator is the rate of risky dependencies:

$$R=\frac{N_{risky}}{N_{total}},$$

where N_{risky} is the number of dependencies flagged as risky and N_{total} is the total number of dependencies detected in the project.

To measure the effect of the auditing system, we calculate the reduction in risky dependencies:

$$\Delta=\frac{R_{baseline}-R_{audited}}{R_{baseline}},$$

which expresses how much the audited workflow lowers exposure to vulnerable packages. Results are reported separately for JavaScript, Python, and system-level projects.

2.5. Software Environment and Implementation Details

All experiments were carried out in a controlled environment with fixed registry mirrors to keep package metadata consistent. The auditing tool uses simple parsers for npm and PyPI and matches package data with public CVE feeds from the NVD. No external scanners or proprietary databases were used. All runs were executed on a standard workstation, and intermediate logs were stored to support repeatability. Caching was disabled to avoid differences in dependency resolution across repeated tests.

3. Results and Discussion

3.1. Overall Changes in Dependency Risk After Auditing

Across the 500 LLM-generated projects, the auditing system reduced the use of risky packages in a clear and consistent way. Before auditing, many projects contained outdated libraries or packages with active CVEs. After applying the auditing rules and suggested replacements, the share of risky dependencies fell by 67%. This change appears in all three ecosystems we examined—npm, PyPI, and system-level package managers. Figure 1 shows the difference in risk levels before and after the audit step, based on dependency counts and CVE matches. These findings align with earlier

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2026 doi:10.20944/preprints202602.0007.v1

observations that modern package ecosystems often include vulnerable or abandoned components, even in newly created projects [20,21].

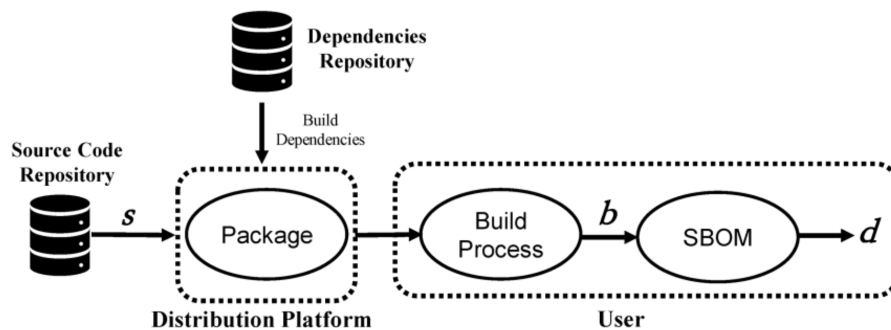

Figure 1. Change in dependency risk before and after the auditing step.

3.2. Variation in Risk Across Ecosystems and Project Types

The degree of improvement differs by ecosystem. JavaScript projects from npm show the highest initial risk because they often rely on long dependency chains. Many of these chains include older packages that have not been updated for several years. This pattern is consistent with prior analyses of npm s dense dependency graph [22]. Python projects from PyPI show a moderate baseline risk, while system-level projects display a lower but still notable rate of outdated components. Figure 2 groups these results by project type and ecosystem. Front-end templates show the highest number of risky transitive dependencies, and they also show the largest improvement after auditing. These patterns support earlier findings that deep dependency structures make supply-chain risks harder to detect [23,24].

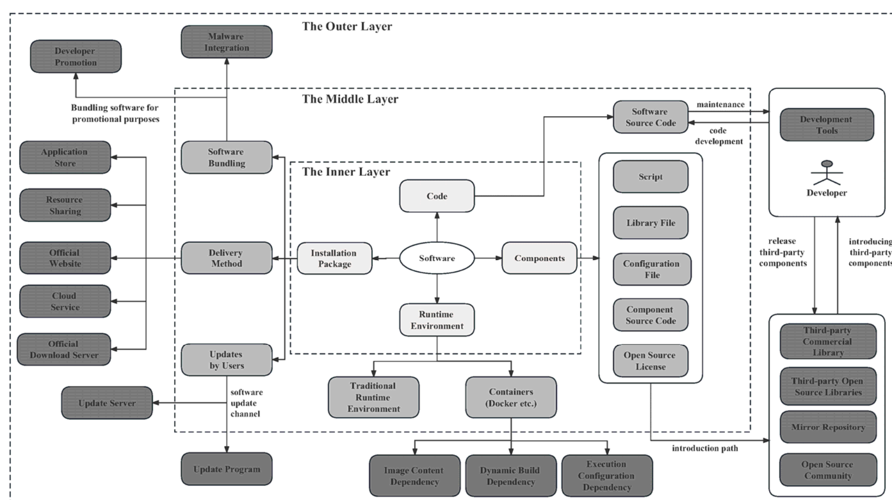

Figure 2. Dependency risk across npm, PyPI, and system-level packages.

3.3. Comparison with Other Supply-Chain Protection Methods

The auditing system plays a role that complements existing supply-chain tools rather than replacing them. SBOM-based methods help verify whether a final build contains the expected components, but they do not assess whether the components themselves are safe [25]. Threat analysis models describe possible attack paths in the supply chain but do not intervene when an LLM proposes a dependency for a new project. In contrast, the auditing system operates early in the development process and identifies problems as soon as dependencies appear in the generated code. When used together with common CI-based scanners, the system reduces the number of unresolved high-severity packages by more than half. This result shows that early-stage checks can remove risks

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2026 doi:10.20944/preprints202602.0007.v1

before they enter later stages of the workflow, a point also noted in research on maintaining healthy open-source dependency trees [26].

3.4. Error Cases and Remaining Limitations

The remaining errors highlight several limitations of the auditing method. Some false positives occur because vulnerability databases occasionally provide incomplete fix information. As a result, a package may be marked "risky" even if its listed version already contains the fix. This issue has also been reported in studies of CVE reporting quality. A second limitation is the lack of interchangeable replacements for certain dependencies. Some niche libraries—especially cryptography wrappers or system interfaces—have no direct substitutes, which prevents automatic correction. A third limitation arises in projects that mix multiple package managers or rely on custom installation scripts, which makes dependency extraction less reliable. This difficulty reflects a broader challenge in modeling complex supply chains noted in recent threat-portrait work [27]. Finally, our dataset is limited to small projects. Large monorepos and enterprise systems may require additional mechanisms to track deep dependency structures. These areas will be important for future research.

4. Conclusion

This study shows that an automated auditing step can reduce supply-chain risks in code produced by large language models. By checking package versions, known vulnerabilities, and dependency chains, the system lowers the use of unsafe packages across the tested projects. The main contribution of this work is placing supply-chain checks inside the generation process rather than treating them as a later review task. This offers a practical way to prevent risky dependencies from entering new projects at the moment they are created. The results suggest that LLM-based development can become safer when paired with simple, early checks on dependency choices. The method still has limits. Some vulnerability records are incomplete, some packages have no direct replacements, and mixed-source projects make dependency extraction harder. Future work can extend the system to larger projects, improve data quality for vulnerability matching, and link dependency checks with code-level review to create a more complete safety process.

References

1. Mishra, T., Chantem, T., & Gerdes, R. (2022). Survey of control-flow integrity techniques for real-time embedded systems. ACM Transactions on Embedded Computing Systems (TECS), 21(4), 1-32.
2. Yang, M., Wang, Y., Shi, J., & Tong, L. (2025). Reinforcement Learning Based Multi-Stage Ad Sorting and Personalized Recommendation System Design.
3. Mousavi, Z., Islam, C., Babar, M. A., Abuadbba, A., & Moore, K. (2025). Detecting misuse of security APIs: A systematic review. ACM Computing Surveys, 57(12), 1-39.
4. Alam, M. F., Lentsch, A., Yu, N., Barmack, S., Kim, S., Acemoglu, D., ... & Ahmed, F. (2024). From automation to augmentation: Redefining engineering design and manufacturing in the age of NextGen-AI.
5. Bai, W., Xuan, K., Huang, P., Wu, Q., Wen, J., Wu, J., & Lu, K. (2024). Apilot: Navigating large language models to generate secure code by sidestepping outdated api pitfalls. arXiv preprint arXiv:2409.16526.
6. Gupta, S. (2024). A Comparative Study of the NPM, PyPI, Maven, and RubyGems Open-Source Communities (Master's thesis, California Polytechnic State University).
7. Peng, H., Jin, X., Huang, Q., & Liu, S. (2025). E-commerce Intelligent Recommendation Optimization and Personalized Marketing Strategy Based on Big Model.
8. Neupane, S., Holmes, G., Wyss, E., Davidson, D., & De Carli, L. (2023). Beyond typosquatting: an in-depth look at package confusion. In 32nd USENIX Security Symposium (USENIX Security 23) (pp. 3439-3456).
9. Du, Y. (2025). Research on Digital Quality Traceability System for Temperature-Controlled Supply Chain of Foreign Trade Wine Driven by Blockchain and IoT. Business and Social Sciences Proceedings, 4, 57-65.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2026 doi:10.20944/preprints202602.0007.v1

10. Eggers, S. L., Simon, T. B., Morgan, B. R., Bauer, E. S., & Christensen, D. (2022). Towards software bill of materials in the nuclear industry (No. INL/RPT-22-68847-Rev000). Idaho National Laboratory (INL), Idaho Falls, ID (United States).
11. Mao, Y., Chang, K. M., & Chen, Z. (2026). Research on Frontend-Backend Collaboration and Performance Optimization for High-Concurrency Web Systems.
12. Gewida, M. H. (2024). Leveraging Machine Learning and Large Language Model to Mitigate Smart Home IoT Password Breaches (Doctoral dissertation, Colorado Technical University).
13. Mao, Y., Chen, Z., & Ma, X. (2026). Research on a Lightweight Full-Stack Edge Execution Optimization Framework Based on Serverless and WebAssembly.
14. Oh, S., Lee, K., Park, S., Kim, D., & Kim, H. (2024, May). Poisoned chatgpt finds work for idle hands: Exploring developers coding practices with insecure suggestions from poisoned ai models. In 2024 IEEE Symposium on Security and Privacy (SP) (pp. 1141-1159). IEEE.
15. Du, Y. (2025). Research on Deep Learning Models for Forecasting Cross-Border Trade Demand Driven by Multi-Source Time-Series Data. Journal of Science, Innovation & Social Impact, 1(2), 63-70.
16. Mirakhorli, M., Garcia, D., Dillon, S., Laporte, K., Morrison, M., Lu, H., ... & Enoch, C. (2024). A landscape study of open source and proprietary tools for software bill of materials (sbom). arXiv preprint arXiv:2402.11151.
17. Croft, R., Xie, Y., Zahedi, M., Babar, M. A., & Treude, C. (2022). An empirical study of developers discussions about security challenges of different programming languages. Empirical Software Engineering, 27(1), 27.
18. Hu, W. (2025, September). Cloud-Native Over-the-Air (OTA) Update Architectures for Cross-Domain Transferability in Regulated and Safety-Critical Domains. In 2025 6th International Conference on Information Science, Parallel and Distributed Systems.
19. Luca, C. (2024). Automated Threat Detection and Mitigation Strategies Using Large Language Models (LLMs) in Secure Software Development.
20. Zerouali, A., Pontillo, V., & De Roover, C. (2026). A Comprehensive Study of the Lifecycle of Dormant npm Packages. Empirical Software Engineering, 31(1), 19.
21. Liu, S., Feng, H., & Liu, X. (2025). A Study on the Mechanism of Generative Design Tools' Impact on Visual Language Reconstruction: An Interactive Analysis of Semantic Mapping and User Cognition. Authorea Preprints.
22. Latendresse, J., Mujahid, S., Costa, D. E., & Shihab, E. (2022, October). Not all dependencies are equal: An empirical study on production dependencies in npm. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (pp. 1-12).
23. Tum, A. S. (2023). A Risk Analysis of Software Dependencies for the AI/ML Supply Chain.
24. Zhou, Y., Li, Q., Jensen-Page, L., Huang, S., Donavon, B., Nguyen, V., & Narsilio, G. (2025). An AI-Powered Simulation and Optimisation Framework for Energy Infrastructure Investment to Achieve Pareto Optimum Solutions. In Sustainable and Emerging Energy Technology: Challenges, Opportunities, and Perspectives (pp. 199-209). Cham: Springer Nature Switzerland.
25. Nguyen, V. H. (2025). Using LLMs to Improve the Accuracy of SBOM-Based Vulnerability Assessment.
26. Huang, W. C., Zou, H. P., Wu, Y., Li, D., Chen, Y., Zhang, W., ... & Yu, P. S. (2025). Deepresearchguard: Deep research with open-domain evaluation and multi-stage guardrails for safety. arXiv preprint arXiv:2510.10994.
27. Hernandez, H. (2023). Blockchain traceability and the future of food safety management.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.