

Agentic Dependency Management at Repository-Level

KAYA KITADA^{1,a)} YOSHIKI HIGO^{1,b)} RAULA GAIKOVINA KULA^{1,c)}

Abstract: Software infrastructure is built upon a complex web of open-source software dependencies, known as library ecosystems. While these dependencies are essential for accelerating delivery, they can also create overreliance, leading to maintenance burdens such as frequent version updates, security vulnerabilities, and supply-chain risks. Developers have traditionally relied on open-source community-led maintenance to mitigate these burdens. However, the emergence of Agentic Software Engineering offers a promising alternative for dependency management. This paper explores whether AI agents can discover repository-level dependencies and assist in replacing selected dependencies with locally generated code. We conduct an initial repository-scale experiment on the real-world Flask open-source repository. We evaluate the agent’s ability to (i) discover dependencies under different visibility conditions and (ii) reimplement dependencies through an incremental removal-and-repair workflow.

Keywords: Agentic Software Engineering, Dependency Management, Open Source Software, Repository-level Automation

1. Introduction

Software engineering has become characterized by an unprecedented reliance on complex ecosystems of external libraries and frameworks. Examples include the NPM^{*1}, PyPI^{*2}, and Maven^{*3} ecosystems that serve millions of applications worldwide. These dependencies act as the fundamental building blocks of modern infrastructure, allowing developers to accelerate delivery cycles by leveraging pre-existing, highly optimized codebases. From web frameworks to specialized machine learning libraries, the ability to integrate third-party components has become a cornerstone of modern software development. A 2024 Synopsys report^{*4} showed that 96% of the commercial codebases they sampled contained open-source software, and 77% of the code within those codebases was open source.

However, this reliance introduces significant, often hidden, costs known as “dependency technical debt.” As software ecosystems evolve, the very libraries that enable rapid development also introduce frequent versioning conflicts, security vulnerabilities, and supply-chain risks[1], [4], [8]. The maintenance burden of keeping dependencies up-to-date and auditing them for malicious code has become a critical challenge[10]. This overreliance occurs when the cost of managing an external dependency—in terms of both developer effort and systemic risk—outweighs the functional benefit it provides to the local codebase. This issue is especially prevalent when libraries are maintained by volunteer

open-source developers who may lack the time or resources necessary to sustain long-term support. One notable example is the Log4Shell vulnerability in the Log4j library^{*5}, where a small but widely adopted logging library integrated into millions of devices worldwide received the highest possible severity score.

Prior work[6], [9] has explored various approaches for understanding and managing software dependencies automatically. However, challenges range from continuously scanning for vulnerabilities to handling version updates, project abandonment, and license inconsistencies[5]. Industry tools such as Dependabot provide automated notifications and support for dependency maintenance, helping developers react to ecosystem changes more efficiently.

The emergence of Agentic Software Engineering[2] presents a transformative opportunity to address these challenges. Large Language Models (LLMs) and autonomous software agents now possess the capability to understand, parse, and generate complex code structures with high fidelity^{*6}. Rather than being immutable external entities, we are beginning to shift toward a paradigm in which agents actively manage the boundary between internal and external code. There has been some initial work on Large Language Models for dependency-related tasks[7], but repository-level dependency management by AI agents remains underexplored. Moreover, while these agents can reduce coding effort, they are a double-edged sword: they may produce AI slop, or a “large quantity of low-quality—and oftentimes inaccurate—contributions that don’t add value to the project.”^{*7}

¹ The University of Osaka, Suita, Osaka 565–0871, Japan

^{a)} ka-kitad@ist.osaka-u.ac.jp

^{b)} higo@ist.osaka-u.ac.jp

^{c)} raula-k@ist.osaka-u.ac.jp

^{*1} <https://www.npmjs.com/>

^{*2} <https://pypi.org/>

^{*3} <https://maven.apache.org/>

^{*4} <https://www.blackduck.com/resources/analyst-reports/open-source-security-risk-analysis.html#introMenu>

^{*5} <https://github.blog/open-source/inside-the-breach-that-broke-the-internet-the-untold-story-of-log4j/>

^{*6} <https://github.blog/news-insights/octoverse/typescript-python-and-the-ai-feedback-loop-changing-software-development/>

^{*7} <https://github.blog/open-source/maintainers/what-to-expect-for-open-source-in-2026/#h-the-double-edged-sword-of-ai>

In this technical report, we take initial steps toward using agentic systems to manage dependency over-reliance in real-world software projects. We emphasize that agentic dependency management in a software engineering context has not yet been sufficiently studied, and that this research is timely in the era of generative AI. Recent work has explored the use of AI agents for dependency management, although much of this research relies on benchmark datasets like the SWE-Bench[3] for standardized evaluation. In contrast, we adopt a more practical SE approach by applying dependency management directly to a real open-source project. To evaluate the feasibility of this approach, we conduct an empirical study at repository scale.

Our case study is the popular Flask^{*8} open-source project, which has 732 developers and over 71 thousand GitHub stars. Using this real-world repository, we experiment with OpenAI ChatGPT 5.4^{*9} as the agent model available at the time of experimentation. Our research focuses on two primary questions:

- **RQ1 (Autonomous Discovery):** To what extent can AI agents identify repository-level dependency candidates under different dependency-visibility conditions? *The motivation is to explore how agents report dependency candidates and identify the software artifacts used as evidence for repository-scale dependency analysis.*
- **RQ2 (Reimplementations):** To what extent can AI agents reimplement library functionality locally, compile the repository, and pass tests after dependency removal? *While RQ1 examines how an agent discovers dependencies, RQ2 investigates whether an AI agent can go one step further and transform an existing repository toward a reduced-dependency implementation.*

The rest of this paper is organized as follows. Section 2 presents the empirical design, including the target repository and the two experimental settings. Section 3 reports the results for autonomous dependency discovery and dependency reimplementations. Section 4 discusses the implications of these findings. Section 5 summarizes limitations and threats to validity. Section 6 outlines future work, and Section 7 concludes the paper.

2. Empirical Design

In this section, we present our dataset and the experimental design for the two research questions.

2.1 Target Repository: Flask

2.1.1 Rationale

In this study, we use the Flask Python web framework. The inclusion of Flask is motivated by two factors, and allows us to cover a broader development context such as tests, documentation, examples, and CI/CD pipelines. First, Flask is a mature and pervasive project, serving as an excellent representative of modern open-source software. At the time of our study, Flask had 236 files, a CI pipeline, and a non-trivial repository structure. Second, Flask has an active OSS community. According to GitHub, Flask had attracted 732 contributors, 5,537 commits, and 2,791 pull requests as of May 16, 2026.

Our collection scope encompasses a comprehensive set of artifacts extracted from the Flask repository. The data collection was performed by capturing a fixed repository snapshot used throughout the experiments. We primarily focus on the core source code, import statements, module references, tests, documentation, examples, and configuration files that indicate internal and external dependencies.

2.1.2 Tools and Environment

To measure the agent interactions, we created a log plugin extension to collect the activities of the agent. We use the open-source pi coding harness^{*10} to execute the agents. Other commercial harnesses, such as Claude Code and Codex, did not provide the same level of customization for creating our own logging plugin extensions. Our logs capture the agent’s autonomous traversal strategy: which files it chose to open, how it reasoned through a specific import, and the moment it concluded that a particular trace constituted a dependency. The experimental period took around two weeks, as we experimented with different configuration setups, resulting in a total cost of approximately 300 USD. Due to size limitations, the current submission includes scripts only; we plan to release the raw logs after camera-ready preparation.

2.2 Approach for RQ1: Autonomous Dependency Discovery

2.2.1 Design

In contrast to related work, we employ an AI agent that is free to decide which repository artifacts to inspect. Our approach does not provide the agent with a predefined search path. Instead, we empower an autonomous agent to navigate the repository architecture independently. For RQ1, we create an agent that is tasked with reading our prompt file and following the instruction below. “List up all third-party dependencies.”. Other guardrails, such as system prompts, were placed in the root folder and will be included in the supplementary materials. By using such an underspecified instruction, we force the agent to perform its own discovery process—deciding which files are worth inspecting, how to interpret their contents, and how to traverse the file hierarchy to find evidence of external software. To capture the broader context of dependency usage, we also incorporate information from testing and documentation assets, including test suites in `tests/`, usage examples found in `docs/` and `README` files, and sample implementations within the `examples/` directory. We compare the agent’s discovery capability under two situations:

- **Declarative-Driven:** The agent operates in a “high-visibility” environment where explicit dependency metadata is readily available. Files such as `pyproject.toml`, `uv.lock`, and `requirements.txt` are present, allowing the agent to rely on declarative, authoritative sources. Hence, the agent has access to documentation, configuration files, and dependency metadata as descriptive text in documentation, and as configuration-based references.
- **Trace-Driven:** We strip the repository of all such declarative metadata. The agent is forced to rely entirely on the “traces” left behind in the codebase—such as import statements in

^{*8} <https://github.com/pallets/flask>

^{*9} <https://openai.com/index/introducing-gpt-5-4/>

^{*10} <https://github.com/earendil-works/pi>

src/, test configurations in tests/, usage examples in docs/, and workflow definitions in .github/workflows/. This setup allows us to measure whether an agent can successfully reconstruct a dependency landscape using only indirect, inferable evidence embedded within the functional components.

2.2.2 Metrics

The experimental procedure involves executing the agent in two isolated sessions. We record the resulting dependency lists (*Declarative* and *Trace*). To compare the two situations, we measure how the AI agent executes (i.e., execution time, dialogue turns, large language model provider requests, the number of tool executions, input tokens and output tokens.) Furthermore, using tool-execution logs for `grep`, `ls`, `find`, `read`, and `write`, we compare which artifacts the AI agent referenced and what exploration steps it took to construct the dependency list.

2.3 Approach for RQ2: Dependency Re-implementation

2.3.1 Design

Algorithm 1 presents the incremental strategy used in this study. Rather than asking the agent to rewrite the entire repository in a single pass, we use an *Incremental Sweep* in which the agent removes one dependency at a time. This staged design allows us to observe where the transformation succeeds, where it fails, what repairs are required, and how quality indicators change across the run. Each run begins from the fixed repository snapshot described above, and before any dependency removal is attempted, we record a baseline consisting of the original dependency inventory, build status, test status, and relevant behavioral observations. After each dependency run, the human supervisor pressed “Continue” to proceed to the next dependency.

The prompt serves as the top-level orchestration prompt given to the AI at the start of the run. Its roles include run initialization, dependency discovery, target dependency set management, step progression control, final validation, and whole-run completion judgment. In short, it acts as the control tower for the entire experiment. This top-level prompt also determines when to invoke Prompt A and Prompt B. The reimplementation is controlled by two prompt types. Prompt A (Dependency Removal) instructs the agent to remove a target dependency without adding new third-party dependencies, reimplement required functionality locally when needed, update or add tests where appropriate, and validate the result through build and test commands. If validation fails within a step, the agent first attempts localized repair as part of the same dependency-removal process. Prompt B (Repair) is invoked only when additional repair is required after build, test, or final validation failures. This separation allows us to distinguish the cost of initial removal from the cost of subsequent repair. Other guardrails, such as system prompts, were placed in the root folder and will be included in the supplementary materials. In the second run, the agent is tasked to repair (Prompt B) iteratively in a recursive process until there is a timeout or all reported removal targets reach a build- and test-passing state.

2.3.2 Metrics

The experimental procedure involves executing the agent in different sessions from RQ1 (*Declarative* and *Trace*). Further-

Table 1 Comparison of Execution and Usage Logs

Metrics	Declarative-Driven	Trace-Driven
Execution time (# sec)	247.4	266.0
# Dialogue Turns	10	24
# Provider Requests	10	24
# Tool Executions	45	97
# Input Tokens	153,323	70,178
# Output Tokens	17,614	16,146

Table 2 Comparison of Tool Usage Frequency

Tool	Declarative-Driven	Trace-Driven
<code>read</code>	31	29
<code>grep</code>	11	45
<code>ls</code>	1	12
<code>find</code>	1	10
<code>write</code>	1	1

more, we measure the effectiveness of the agent by distinguishing between first-run success and second-run repair success. Similar to RQ1, we measure how the AI agent executes (i.e., execution time, dialogue turns, large language model provider requests, the number of tool executions, input tokens and output tokens).

3. Results

In this section, we present the results corresponding to both research questions.

3.1 Answering RQ1

Interestingly, we find that the agent was inconsistent in its detection. For the declarative-driven approach, the agent reported 114 dependencies, compared to 28 by the trace-driven approach. Note that we did not verify this number against an independent ground truth. Instead, the first author manually checked the declared configuration files as a metadata-based reference. We do not treat this metadata as a complete ground truth, because declarative files may include optional, test, documentation, tooling, or transitive dependencies, and may not fully capture trace-level dependency usage. Validation against a complete ground truth is a much trickier problem, which we discuss in the limitations. As shown in Table 1 and Table 2, we find differences in the agent activities. Our execution logs show that metadata availability changed not only the final dependency list but also the agent’s search behavior. The declarative-driven approach completed in 10 turns with 45 tool executions, whereas the metadata-absent condition required 24 turns and 97 tool executions. The metadata-absent condition also used substantially more repository-wide search operations, especially `grep`, `ls`, and `find`, indicating that the agent first had to locate evidence before interpreting it.

Summary for RQ1

We find that declarative metadata leads the agent to report a broader set of dependency candidates. However, trace-driven discovery requires more exploratory tool use because the agent must first locate evidence before interpreting it.

3.2 Answering RQ2

Note that the experiments for RQ1 and RQ2 are independent;

Algorithm 1 Two-phase incremental sweep used for dependency reimplementa-

Require: List of dependencies D **Ensure:** Build- and test-passing states for reported removal targets

```
1: Human supervisor starts the run with the orchestration prompt
2:  $Failed \leftarrow \emptyset$ 
3: for all  $d \in D$  do
4:   Reimplement  $d$  using Prompt A
5:   if  $d$  fails then
6:     Repair  $d$  once using Prompt B
7:   end if
8:   if  $d$  still fails then
9:     Add  $d$  to  $Failed$ 
10:  end if
11:  Human supervisor presses "Continue" for the next dependency
12: end for
13:  $FailCount \leftarrow |Failed|$ 
14: End first run; human supervisor starts the second run
15: while  $Failed \neq \emptyset$  do
16:    $Current \leftarrow Failed$ 
17:    $Failed \leftarrow \emptyset$ 
18:   for all  $d \in Current$  do
19:     Reimplement or repair  $d$ 
20:     if  $d$  fails then
21:       Add  $d$  to  $Failed$ 
22:     end if
23:     Human supervisor presses "Continue" for the next dependency
24:   end for
25: end while
26: return  $FailCount$ 
```

Table 3 Successful coding, test, and build rate after the first run for RQ2.

Condition	Success	Failed
Declarative-Driven	46.5%	53.5%
Trace-Driven	81.0%	19.0%

Table 4 Comparison after the two-phase Incremental Sweep.

Metric	Declarative-Driven	Trace-Driven
Total Execution Time (hours)	3.97	1.40
# Harness runs	54	23
# Dialogue Turns	1068	403
# Input tokens	2,532,419	693,978
# Output tokens	759,333	324,800

Table 5 Additional agent workload required after the first run until the second run was completed.

	Declarative-Driven	Trace-Driven
Execution time (# hours)	1.08	0.23
# Targets requiring repair	23	4
# Harness Runs	9	2
# Turns	310	53
# Tool executions	528	90
# Input tokens	615,776	61,019
# Output tokens	195,307	54,474

therefore, we do not use any dependencies identified in RQ1. Because RQ2 uses a separate agent session with an additional dependency-removal instruction, the dependency sets reported in RQ2 are not identical to the RQ1 discovery outputs. We repeat the discovery process with the additional instruction that the agent should remove the dependency. The declarative-driven approach reported 43 dependency-removal targets, whereas the trace-driven approach reported 21 dependency-removal targets.

Table 3 presents the success rate after the initial execution pass of the incremental sweep. Surprisingly, the Trace-Driven con-

dition achieved a substantially higher first-pass success rate of 81.0%, leaving only 19.0% of targets unresolved after the first incremental sweep. In contrast, the Declarative-Driven condition successfully completed only 46.5% of targets, with the remaining 53.5% requiring additional recursive repair iterations. This suggests that the presence of declarative metadata did not necessarily simplify the reconstruction process, and may instead have introduced additional complexity or ambiguity for the agent during dependency reimplementa-
tion.

Table 4 compares the overall execution metrics after completion of the two-phase repair process. The Declarative-Driven condition required significantly more computational effort across all measured dimensions. Specifically, it executed 54 harness runs and 1,068 dialogue turns, compared to only 23 runs and 403 turns in the Trace-Driven setting. Similarly, the Declarative-Driven workflow consumed nearly three times longer execution duration (3.97 hours versus 1.40 hours), as well as substantially higher token usage, totaling over 2.5 million input tokens and 759 thousand output tokens. In comparison, the Trace-Driven condition required only 694 thousand input tokens and 325 thousand output tokens. These results indicate that recursive repair cycles imposed a substantially larger workload in the Declarative-Driven environment.

Table 5 further isolates the additional workload introduced during the recursive repair phase after the first run. The Declarative-Driven condition required reprocessing 23 additional targets, resulting in 9 extra harness executions, 310 additional dialogue turns, and 528 more tool calls. This recursive recovery phase added 1.08 hours of execution time and consumed an additional 615,776 input tokens and 195,307 output tokens. In contrast, the Trace-Driven condition required repair for only 4 targets, incur-

ring minimal overhead with just 2 additional runs, 53 turns, and 90 tool calls.

Summary for RQ2

We find that in both situations, the AI agent eventually produced build- and test-passing repository states for all reported dependency-removal targets. Surprisingly, the agentic workflows are noticeably different, with the declarative-driven condition requiring more tokens and more agent activity. In both situations, the agent was not fully successful in the first run.

4. Discussion

In this section, we discuss what the Flask results imply for agentic dependency discovery, dependency removal, and evaluation design.

4.1 Dependency visibility changes the agent's scope.

The Flask case study shows that dependency discovery by an AI agent is strongly shaped by the artifacts that are visible to the agent. In the declarative-driven condition, the agent reported 114 dependencies, whereas in the trace-driven condition it reported 28. This difference should not be interpreted only as one condition being more accurate than the other. Rather, declarative metadata exposed a broader dependency surface, including direct dependencies, transitive dependencies, example-specific dependencies, and documentation or deployment-related dependencies. In contrast, the trace-driven condition converged on a narrower set of dependencies that were more visible from source, tests, documentation traces, and examples. This suggests that agentic dependency discovery is sensitive to what the agent treats as authoritative evidence.

4.2 Dependency discovery is goal-sensitive.

Dependency discovery is also shaped by the downstream goal given to the agent. In RQ1, the agent was asked to list third-party dependencies, whereas in RQ2 the task included dependency removal and local reimplementation. Although both settings required dependency discovery, the reported dependency sets were smaller in RQ2: the declarative-driven condition reported 114 dependencies in RQ1 but 43 removal targets in RQ2, while the trace-driven condition reported 28 dependencies in RQ1 but 21 removal targets in RQ2. This suggests that the agent did not treat dependency discovery as a task-neutral inventory process. Instead, when removal was included in the task, the agent may have implicitly narrowed the target set to dependencies it considered actionable, relevant, or removable. This result indicates that dependency-management agents should be evaluated together with the management goal they are asked to support.

4.3 More dependency information can increase reconstruction cost.

A key finding from RQ2 is that more explicit dependency information did not necessarily make reimplementation easier. The

declarative-driven condition identified more removal targets and achieved a lower first-pass completion rate than the trace-driven condition (46.5% versus 81.0%). It also required more total effort after the two-phase sweep: 54 harness runs, 1,068 dialogue turns, and 3.97 hours, compared with 23 harness runs, 403 dialogue turns, and 1.40 hours in the trace-driven condition. This suggests a trade-off between dependency coverage and agentic execution cost. Rich metadata helps the agent discover a broader dependency landscape, but it may also expand the reconstruction responsibility and increase the number of repair iterations.

4.4 Agentic dependency management is a repair process, not only a generation task.

The experiments indicate that dependency reimplementation should not be evaluated only as a single-pass code generation problem. In both conditions, the agent required repair iterations before all reported removals were completed. The additional repair workload was especially large in the declarative-driven condition, where 23 targets remained after the first run and required 9 additional harness runs, 310 additional dialogue turns, and 528 additional tool calls. This highlights the importance of evaluating agentic workflows by repair efficiency, convergence behavior, and validation strategy, not only by final completion.

4.5 Implications for evaluating dependency-removal agents.

The results suggest that future evaluations should distinguish between dependency discovery, dependency selection, reimplementation, validation, and repair. A dependency list that is broader is not necessarily better if it includes transitive, documentation-only, example-only, or tooling dependencies that are not required for the target runtime behavior. Conversely, a narrower trace-based list may be easier to reconstruct but may miss dependencies that are relevant for documentation, testing, or packaging. For repository-level agents, the evaluation target should therefore be made explicit: removing runtime dependencies, reconstructing test dependencies, reducing supply-chain exposure, or producing a fully self-contained repository are different goals.

Practical Takeaways

For a practical dependency-management agent, our results suggest three design principles. First, dependency discovery should not stop at listing package names; the agent should classify each dependency by evidence type and role, such as runtime, optional, test, documentation, build, workflow, transitive, or example-only. Second, dependency removal should be incremental and validated after each step, because failures often emerge only after a replacement interacts with tests, configuration, or packaging behavior. Third, the agent should preserve a repair log that records what was removed, what local replacement was introduced, which validation commands were executed, and why additional repair was needed.

5. Limitations and Threats to Validity

In this section, we summarize the main limitations and threats to validity. As this is emerging work, we acknowledge that stronger claims require a larger and more systematic study. We now summarize five different threats to the study and how we mitigated and setup for future work.

5.1 Single-repository scope.

This study focuses on Flask as an initial repository-scale case study. Flask is a useful target because it contains runtime dependencies, optional integrations, tests, documentation, examples, and CI/CD configuration. However, dependency structures differ substantially across ecosystems and project types. A command-line tool, a data-science library, a compiler toolchain, or a JavaScript frontend may expose different dependency traces and different reimplementations challenges. Therefore, the current results should be interpreted as a feasibility study rather than a general claim about all repositories.

5.2 Ground-truth ambiguity.

At this stage, we report the outputs provided by the AI and do not yet have a complete ground truth. Based on manual inspection of the declared configuration files as a metadata-based reference, we find that the declarative-driven approach appears optimistic and tends to overestimate dependencies. This is not simply an agent error: the definition of a dependency is itself ambiguous. A package can be a direct runtime dependency, a transitive lock-file dependency, a documentation dependency, a test dependency, a pre-commit tool, a workflow tool, or an example-only package. Future work should create a labeled dependency taxonomy so that precision and recall can be computed separately for each category.

5.3 Validation limits.

This study primarily relies on build and test completion as validation signals. Passing tests is important, but it does not prove semantic equivalence with the original dependency-based implementation. Generated replacements may behave differently for untested inputs, edge cases, exception types, performance-sensitive paths, security properties, or API compatibility. A stronger evaluation should compare original and generated implementations using behavioral equivalence tests, mutation-based test generation, differential testing, and manual review of high-risk replacements.

5.4 Model and harness dependence.

The experiment used a single agent/model setting based on OpenAI ChatGPT 5.4 and the pi coding harness. Different models may explore repositories differently, produce different replacement strategies, or fail in different ways. Similarly, harness design, available tools, context-window behavior, and logging overhead may affect the number of turns, tool calls, tokens, and repair iterations. Future work will evaluate additional closed-source and open-source models, alternative harness configura-

tions, and repeated trials to estimate variance.

5.5 Human-in-the-loop effects.

Our RQ2 workflow includes human supervision between dependency-removal steps: the human operator had to continue the run after each dependency. This design helped control the experiment and inspect intermediate progress, but it also means the workflow is not fully autonomous. Future experiments should separate fully autonomous execution from supervised execution and measure how human checkpoints affect success, cost, and failure recovery.

6. Future Work

In this section, we outline the next research steps that follow from the Flask experiment. We are optimistic that agentic solutions can help us understand the long-term trade-off between dependency coverage, reconstruction cost, validation effort, and maintainability.

In the subsequent sections, we now list up 10 different research directions and points that will be looking into for future work. This involves repository understanding, ground-truth, builds, scaling and so on.

6.1 Repository understanding for dependency discovery.

A promising direction for RQ1 is to make the agent’s understanding of repository structure more systematic. In the current experiment, the agent inferred dependencies by reading metadata files, import statements, tests, documentation, examples, and workflow files. This makes the result sensitive to the agent’s ad hoc search behavior. Future work should investigate how richer repository understanding can help the agent interpret where dependencies are used, which project activities they support, and whether they are relevant to runtime behavior, testing, documentation, workflows, transitive resolution, or examples. This direction would make RQ1 less dependent on search behavior and would provide stronger evidence for constructing a dependency ground truth.

Future work should also make dependency discovery goal-aware. Our results suggest that the dependency set reported by an agent can change when the task shifts from inventory construction to dependency removal and reimplementations. Future dependency-management agents should therefore distinguish dependencies that are listed for inventory purposes from those that are actionable targets for auditing, updating, removal, or reimplementations. They should also expose why each dependency is selected or excluded for a given management goal.

6.2 Ground-truth construction and dependency taxonomy.

Future work should construct a manually validated ground truth for RQ1 before making strong claims about detection performance. This ground truth should not be a single flat list of package names. Instead, it should label each dependency by role, evidence source, directness, and expected management goal. For example, runtime dependencies, test dependencies, documentation dependencies, workflow tools, transitive dependencies, and example-only dependencies should be evaluated separately.

This would allow future studies to report precision and recall for each dependency category and to determine whether an agent is over-reporting low-relevance dependencies or missing high-impact runtime dependencies.

6.3 Functional equivalence beyond build success.

The current experiment uses build and test completion as practical validation signals. However, passing tests does not guarantee semantic equivalence with the original dependency-based implementation. Future work should evaluate generated replacements using API-level behavior, exception behavior, edge cases, input-output equivalence, compatibility with downstream usage, and maintainability of the generated local code.

6.4 Scaling beyond Flask and beyond one model.

This study intentionally focuses on Flask as an initial repository-scale case study. Future work should repeat the experiment across repositories with different sizes, domains, dependency structures, and test coverage levels. The current experiment also uses one agent/model setting, so future studies should compare multiple LLMs, agent harnesses, prompting strategies, and context-management approaches.

6.5 Separating useful reconstruction from operational overhead.

The logs suggest that a large part of agentic cost can arise from tracking, recording, validation, and repair rather than from code editing alone. Future work should separate code-changing actions from bookkeeping actions, validation actions, and search actions. This would make it possible to estimate which parts of an agentic dependency-management workflow are essential to correctness and which parts are overhead introduced by the experimental harness or logging design.

6.6 Understanding Recursive Repair Complexity

Future work should investigate what increases recursive repair complexity, which was observed in RQ2. Although the declarative-driven condition successfully recovered all dependencies, it unexpectedly produced lower first-pass success rates and substantially higher computational costs. Understanding how agents interpret and prioritize declarative metadata represents an important direction for improving dependency-aware software agents. In particular, future studies should examine when declarative metadata helps dependency resolution and when it introduces ambiguity, version conflicts, or over-constrained reconstruction behavior.

6.7 Agentic Repair Workflows

RQ2 results highlight the importance of *agentic repair workflows*. In both conditions, completing all reported removal targets required recursive repair iterations rather than a purely single-pass generation process. The majority of computational resources were consumed during recovery and repair stages, not during initial synthesis. Future work should therefore study repair efficiency, recovery strategies, failure localization, and convergence behavior over iterative executions. This would make it possible to

evaluate agents not only by whether they eventually succeed, but also by how efficiently and reliably they recover from failures.

6.8 Agentic costs

Future work should investigate *cost-aware software agents*. The substantial differences in token consumption, execution time, and tool invocation frequency suggest that dependency reconstruction has significant operational costs. Future work may explore adaptive planning strategies, caching mechanisms, retrieval optimization, or lightweight repository summarization techniques that reduce search overhead while preserving reconstruction quality. It should also examine the trade-off between agent efficiency and the correctness of reconstructed dependencies.

6.9 AI-artifacts

RQ2 findings motivate a broader investigation into AI-related artifacts. Modern software development increasingly incorporates prompts, retrieval systems, workflow scripts, skill definitions, and generated context as operational specifications. Future studies should examine how these emerging artifacts contribute to software reconstruction, intent preservation, and automated maintenance. In particular, understanding the interaction between declarative specifications and trace-based evidence may help define new hybrid forms of software documentation optimized for agents.

6.10 Human-in-the-loop

Finally, future work should focus on *human intent alignment and verification*. Although the agents successfully produced build- and test-passing repository states, this does not guarantee semantic correctness, maintainability, or alignment with developer expectations. This opens a broader research challenge on how to validate generated software against human intent, architectural constraints, and long-term maintainability costs and trade-offs. Future studies should examine how testing, behavioral verification, developer feedback, and specification mining can be incorporated into agentic workflows.

7. Conclusion

This Flask feasibility study shows that repository-level dependency management can be partially supported by agentic workflows, but that such workflows are sensitive to dependency visibility and management goals. As agentic systems improve in reasoning, code generation, retrieval, and autonomous execution, dependency management may shift from a largely manual maintenance activity toward a more adaptive process integrated into the software development lifecycle. At the same time, our results show that richer dependency metadata can broaden the agent’s action space and substantially increase repair cost. This highlights the need for dependency-role classification, incremental validation, and auditable repair logs when evaluating repository-level dependency-management agents.

Acknowledgments This work was supported by JSPS KAKENHI Grant Number JP24H00692, JP23K28065, JP25K03102, JP26K02889, and JP26H02500.

References

- [1] Cox, R.: Surviving software dependencies, *Commun. ACM*, Vol. 62, No. 9, p. 36–43 (online), DOI: 10.1145/3347446 (2019).
- [2] Hassan, A. E., Li, H., Lin, D., Adams, B., Chen, T.-H., Kashiwa, Y. and Qiu, D.: Agentic Software Engineering: Foundational Pillars and a Research Roadmap (2025).
- [3] Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O. and Narasimhan, K.: SWE-bench: Can Language Models Resolve Real-World GitHub Issues? (2024).
- [4] Kula, R. G., German, D. M., Ouni, A., Ishio, T. and Inoue, K.: Do developers update their library dependencies?, *Empirical Softw. Engg.*, Vol. 23, No. 1, p. 384–417 (online), DOI: 10.1007/s10664-017-9521-5 (2018).
- [5] Mens, T. and Decan, A.: An Overview and Catalogue of Dependency Challenges in Open Source Software Package Registries (2024).
- [6] Ossher, J., Bajracharya, S. and Lopes, C.: Automated dependency resolution for open source software, *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pp. 130–140 (online), DOI: 10.1109/MSR.2010.5463346 (2010).
- [7] Tawosi, V., Alamir, S., Liu, X. and Veloso, M.: LLM Agents for Automated Dependency Upgrades, *2025 40th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, IEEE, pp. 34–37 (2025).
- [8] Tsakpinis, A.: Analyzing Maintenance Activities of Software Libraries, *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering, EASE '23*, New York, NY, USA, Association for Computing Machinery, p. 313–318 (online), DOI: 10.1145/3593434.3593474 (2023).
- [9] Wang, P., Yang, J., Tan, L., Kroeger, R. and Morgenthaler, J. D.: Generating precise dependencies for large software, *Proceedings of the 4th International Workshop on Managing Technical Debt, MTD '13*, IEEE Press, p. 47–50 (2013).
- [10] Williams, L., Benedetti, G., Hamer, S., Paramitha, R., Rahman, I., Tamanna, M., Tystahl, G., Zahan, N., Morrison, P., Acar, Y., Cukier, M., Kästner, C., Kapravelos, A., Wormke, D. and Enck, W.: Research Directions in Software Supply Chain Security, *ACM Trans. Softw. Eng. Methodol.*, Vol. 34, No. 5 (online), DOI: 10.1145/3714464 (2025).