DEPSRAG: Towards Agentic Reasoning and Planning for Software Dependency Management

Mohannad Alhanahnah
University of Wisconsin-Madison, USA
mohannad@cs.wisc.edu

Yazan Boshmaf
Qatar Computing Research Institute, Qatar
yboshmaf@hbku.edu.qa

Abstract

In the era of Large Language Models (LLMs) with their advanced capabilities, a unique opportunity arises to develop LLM-based digital assistant tools that can support software developers by facilitating comprehensive reasoning about software dependencies and open-source libraries before importing them. This reasoning process is daunting, mandating multiple specialized tools and dedicated expertise, each focusing on distinct aspects (e.g., security analysis tools may overlook design flaws such as circular dependencies, which hinder software maintainability) and creating a significant bottleneck in the software development lifecycle.

In this paper, we introduce DEPSRAG, a multi-agent framework designed to assist developers in reasoning about software dependencies. DEPSRAG first constructs a comprehensive Knowledge Graph (KG) that includes direct and transitive dependencies. Developers can interact with DEPSRAG through a conversational interface, posing queries about the dependencies. DEPSRAG employs Retrieval-Augmented Generation (RAG) to enhance these queries by retrieving relevant information from the KG and external sources, such as the Web and vulnerability databases, thus demonstrating its adaptability to novel scenarios. DEPSRAG incorporates a Critic-Agent feedback loop to ensure the accuracy and clarity of LLM-generated responses. We evaluated DEPSRAG using GPT-4-Turbo and Llama-3 on three multi-step reasoning tasks, observing a threefold increase in accuracy with the integration of the Critic-Agent mechanism. DEPSRAG demo and implementation are available at: https://github.com/Mohannadcse/DepsRAG.

1 Introduction

The increasing reliance on code reuse in modern software development, facilitated by third-party packages, has substantially enhanced both software quality and productivity [25]. As the adoption of open-source and third-party libraries becomes more widespread, securing the software supply chain has emerged as a critical concern [27]. In response, regulatory measures such as Executive Order 14028 [3] and the 2023 EU Cyber Resilience Act (CRA) [2] impose strict requirements on developers to ensure compliance of software dependencies, including open-source and third-party libraries, within their projects. A recent survey [29] highlights a widespread lack of confidence among developers in the effectiveness of their current dependency management practices. This uncertainty contributes to a prolonged and cumbersome approval process for incorporating new open-source libraries for 61% organizations, thus introducing a critical bottleneck in the software development lifecycle. This challenge is further compounded by the lack of advanced tools capable of thoroughly analyzing and addressing security and maintainability concerns in open-source and third-party libraries. Existing security tools often overlook essential issues, such as detecting circular dependencies or resolving version conflicts. For example, within the Python ecosystem, developers

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

typically rely on tools such as "pip-audit" [1] to perform security checks and "piptree" [2] to visualize dependencies in a hierarchical format. However, detecting circular dependencies often requires manual inspection or the use of additional tools such as "pipdeptree" [3].

To tackle these challenges, we introduce DEPSRAG, an LLM-powered digital assistant designed to facilitate informed decision-making on software dependencies before their integration. Central to DEPSRAG 's architecture is a multi-agent system that incorporates Retrieval-Augmented Generation (RAG) alongside planning and critique agents, enabling in-depth reasoning about direct and transitive dependencies. These dependencies are modeled as a Knowledge Graph (KG). The developer can then interact with DEPSRAG to answer questions such as the number of dependencies, the identification of the key packages, the depth of the graph, and the dependency paths. Furthermore, DEPSRAG is capable of acquiring knowledge from external sources, including the Web and vulnerability databases, to respond to unforeseen queries.

In contrast to conventional tools that depend on predefined queries, DEPSRAG uses LLMs to dynamically generate queries to retrieve information from the KG, significantly enhancing its scalability and adaptability. Using the RAG, DEPSRAG integrates KG data to improve the accuracy and relevance of responses to user questions. To further ensure the precision and reliability of its results, DEPSRAG uses a Critic-Agent interaction, which iteratively refines its responses through reasoning and validation, producing consistently accurate and reliable results.

2 Background

2.1 Dependency Graphs

Dependency graphs are essential in software engineering for representing dependencies among software entities like classes, functions, modules, packages, or larger components. Each node represents an entity and directed edges indicate dependencies, meaning changes in one entity can affect another.

Litzenberger et al. [22] proposed a framework with three levels of dependency granularity: package-to-package, artifact-to-package, and artifact-to-artifact. Düsing and Hermann [15] used artifact-to-package graphs to study vulnerability propagation in software repositories. Benelallam et al. [10] created the Maven Dependency Graph to explore artifact releases, evolution, and usage trends. Dependency graphs are also utilized in tools for program understanding [16, 26].

2.2 Knowledge Graphs (KGs)

Knowledge graphs (KGs) organize information in a structured format, capturing relationships between real-world entities and making them comprehensible to humans and machines [7]. In a KG, data are organized as triplets (head entity, relation, tail entity), where the relation is the relationship between these two entities, such as ("Steven Jobs", "owns", "Apple"). More formally, KGs store structured knowledge as a collection of triples $\mathcal{KG} = \{(e_i, r, e_j) \subseteq \mathcal{E} \times \mathcal{R} \times \mathcal{E}\}$, where $\mathcal{E}$ and $\mathcal{R}$ denote the set of entities and relations, respectively. KGs are created by describing entities and entity relationships, known as graph schema. KGs are useful for a variety of applications, such as question-answering [18], information retrieval [30], recommender systems [33], cybersecurity [6, 31], and natural language processing [32].

2.3 Agentic LLM Applications

Agents. An agent is responsible for a specific aspect of a task, acting essentially as a message transformer. In code, an agent is typically represented as a class encapsulating an interface to an LLM, along with optional tools and external data (e.g., a vector or graph database). Agents communicate by exchanging messages, similar to the actor model in programming [19]. For practical applications, agents can trigger actions (e.g., API calls, computations) and access external data, facilitated by tools and RAG.

[1] https://pypi.org/project/pip-audit/
[2] https://pypi.org/project/piptree/
[3] https://pypi.org/project/pipdeptree/

Tools, also known as functions or plugins. Tools allow LLMs to trigger external actions beyond generating text [13]. While free-form text is useful for descriptions, summaries, or agent queries, structured outputs are needed for actions like API calls, code execution, or database queries. In such cases, the LLM produces a structured response, typically in JSON format, specifying necessary details like code or query parameters. The LLM uses a tool when generating such structured responses, and a tool handler is defined to execute the corresponding action when recognized in the LLM's output.

Retrieval-Augmented Generation (RAG). Utilizing an LLM in isolation presents two primary limitations: (a) its responses are confined to the knowledge available at the time of pre-training, making it incapable of addressing questions related to private sources (such as databases or documents) or post-training information, and (b) it cannot verify the correctness of its responses [13]. RAG mitigates these challenges by generating answers based on specific documents or data while also providing source references for validation. When an LLM receives a query Q in the RAG process, it retrieves a set of k most relevant pieces of data $D = d_1, d_2, \ldots, d_k$ from a database store. The query is then reformulated into a new prompt: "Given the following data: $[d_1, d_2, \ldots, d_k]$, provide an answer to this question: Q, based only on these data, and indicate which data support your answer."

Multi-Agent Orchestration An orchestration mechanism is critical for ensuring task progression, managing message flow, and handling deviations from instructions. This work uses a multi-agent programming approach where agents communicate via message exchange for both inter-agent and intra-agent interactions. Frameworks like Langroid [11] and Langchain [12] provide robust tools for message routing and task delegation. In this work, Langroid is employed, where orchestration is encapsulated in the Task class. This class manages user interactions, tool integration, and sub-task delegation by processing a "current pending message" (CPM) through responders, iterating through steps until task completion [13].

3 Motivating Example and Challenges

In this Section, we present the following critical software dependency task to identify risky dependencies.

> "*For package X, version Y under software ecosystem Z, which packages in X has the most dependencies relying on them, and what is the risk associated with a vulnerability in those packages?*"

We emphasize that developing an AI digital assistant system for managing software dependencies and answering tasks like the above example involves several key challenges: (a) the complex, hierarchical nature of software dependencies, which complicates the conventional application of standard RAG methods; (b) the fragility of LLMs, prone to instruction deviation, hallucination, and inaccurate outputs; (c) decomposing the task into manageable sub-tasks while extracting relevant information from the dependency graph and assessing associated risks by retrieving vulnerability data; (d) aggregating data from various sources to formulate a comprehensive response; (e) efficiently routing and delegating tasks to appropriate agents, while mitigating risks like infinite loops and deadlocks through careful orchestration [13]; and (f) validating the final response and refining the query to enhance retrieval accuracy from the KG.

Therefore, we designed DEPSRAG to handle these complex tasks by decomposing them into sub-tasks as depicted in Figure 1. Accordingly, given a query of this form, DEPSRAG executes the following sub-tasks, given X, Y, and Z:

Sub-Task 1: Construct the dependency graph for the software package X, version Y, and from ecosystem Z. The agent `DependencyGraphAgent` will construct the dependency graph by executing the tool `ConstructKGTool`.

Sub-Task 2: Determine nodes that have the highest in-degree in the dependency graph. The agent `DependencyGraphAgent` will translate this question into a query to retrieve from the dependency graph the highest in-degree nodes by executing the tools `GraphSchemaTool` and `CypherQueryTool`.

Sub-Task 3: Query the vulnerability database for each node obtained in Sub-Task 2 and generate a structured report identifying vulnerable nodes within the dependency graph. The

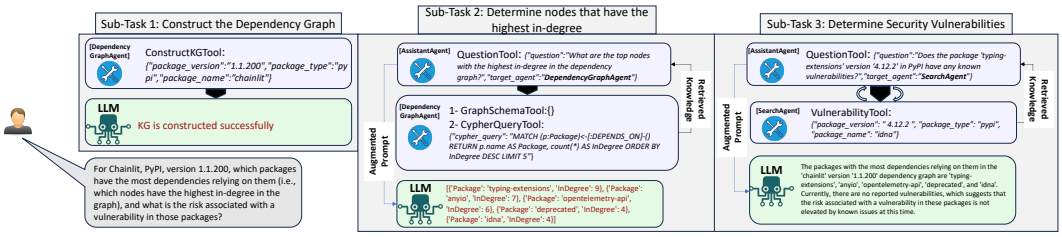

Figure 1: A real-world demonstration of DEPSRAG, applied to identify critical packages and associated risks within the dependency graph of the `Chainlit` package version 1.1.200. This process involves a sequence of subtasks executed by two agents: `DependencyGraphAgent` and `SearchAgent`. The question in Sub-Task 3 will be repeated for each package received from the LLM response in Sub-Task 2. Therefore, `AssistantAgent` needs to orchestrate the communication between these agents, by setting the field "target_agent" in JSON message generated by `QuestionTool`, aggregate the responses received in Sub-Task 2 and Sub-Task 3, and subsequently forward the agents' responses to the `CriticAgent` for validation (omitted in this Figure for the sake of simplicity).

> `SearchAgent` leverages the `VulnerabilityTool` to query the vulnerability database, returning vulnerability results for each node to the `AssistantAgent`, which subsequently consolidates the findings into a structured report.

Both `DependencyGraphAgent` and `SearchAgent` can be paired with a `CriticAgent` that evaluates and provides feedback on the outputs generated by the primary agent (i.e., `AssistantAgent`). Based on this feedback, the primary agent iteratively refines its response. This Critic-Agent loop continues until the Critic agent endorses the final output. This interaction paradigm substantially improves the reliability of DEPSRAG. Section 4 introduces the details of DEPSRAG and discusses in detail the Critic-Agent interaction.

4 DEPSRAG: Multi-Agent System for Dependency Management

This section presents the architecture of DEPSRAG, a chatbot designed for the automated construction of software dependencies as a KG and the subsequent answering of user queries, expressed in natural language, regarding the dependency graph.

4.1 DEPSRAG Critic-Agent Interaction and Orchestration Architecture

This is the core multi-agent interaction pattern that underlies DEPSRAG, and is reminiscent of Actor/Critic methods in reinforcement learning [20].

In an Agent-Critic framework, the Agent serves as the principal entity responsible for processing external inputs and generating output. It is tasked with achieving a specific goal, is equipped with instructions, and has access to the necessary tools and resources. In our implementation, the agent's objective typically involves specialized question-answering. The available resources may include data sources, other agents, or multi-agent systems, while the tools consist of structured mechanisms to invoke API calls, query databases, or perform computations. The Agent's primary role is to formulate a sequence of queries to these resources in pursuit of its assigned goal. It is required to generate a semi-structured message that includes its proposed solution, the reasoning process leading to this solution, and a justifica-

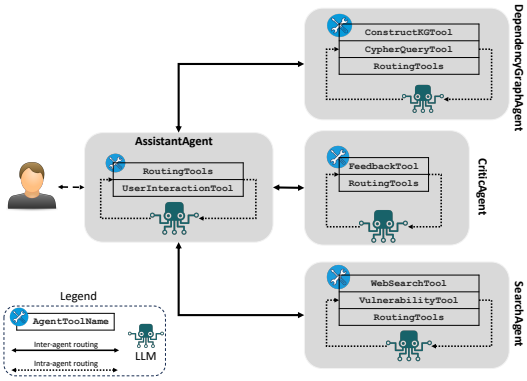

Figure 2: DEPSRAG high-level orchestration architecture

tion—citing relevant sources where applicable. This message is then submitted for evaluation and feedback from the Critic.

To this end, DEPSRAG comprises four agents: `AssistantAgent`, `DependencyGraphAgent`, `SearchAgent`, and `CriticAgent`, each designed to enhance the prompt with supplementary information, as illustrated in Figure 2. Table 2 lists the tools associated with each agent. The `AssistantAgent` coordinates the interactions among these agents by decomposing complex tasks into simpler sub-tasks, delegating them to the appropriate agent, and consulting the `CriticAgent` for feedback on the responses generated before delivering the final answer to the user. The following is a description of these agents:

- `AssistantAgent`. This agent manages the interaction between the user and multiple agents. It further decomposes complex queries into simpler sub-queries, delegates them to the appropriate agents, validates the responses, and aggregates the results into a cohesive answer.
- `CriticAgent`. The Critic is an auxiliary agent paired with the primary Agent described above. Its primary function is to evaluate the Agent's reasoning process, ensuring alignment with the given instructions, and provide constructive feedback. This iterative feedback loop has been demonstrated to enhance the quality of LLM-generated outputs [13, 23]. The Agent refines its response based on the Critic's input, repeating this process until the Critic approves the output, after which the Agent signals task completion and delivers the final results.
- `DependencyGraphAgent`. This agent encompasses a suite of tools designed for constructing, visualizing, and retrieving information from the KG. In particular, for retrieving information from the KG, `DependencyGraphAgent` translates user queries expressed in natural language into the query language of the KG database system, executes the query, and relays the response to `AssistantAgent`. Before accessing the KG, `DependencyGraphAgent` utilizes the `GraphSchemaTool` to retrieve the KG schema, ensuring the query is generated with the correct entity names and relationships. For tasks such as visualization and schema retrieval, we chose to implement queries directly, bypassing the use of LLMs for query generation. This decision aligns with a key design principle in our system, termed the LLM Minimization Principle [13]. According to this principle, tasks that can be deterministically and explicitly defined within standard programming paradigms should be executed directly, without LLMs, to enhance reliability and minimize token usage and latency [13].
- `SearchAgent`. This agent is dedicated to searching for information on the Web or in the vulnerability database. The Web search tool `WebSearchTool` aims to provide answers beyond the scope of the KG and vulnerability database. For example, the user could ask DEPSRAG about the latest version of a specific package.

In DEPSRAG, we developed a suite of advanced routing tools, built on top of the Langroid framework, to enhance reasoning and comprehension within software dependency environments. These extensions are designed to improve the DEPSRAG 's ability to handle complex dependency structures and facilitate more accurate analysis and decision-making processes. All DEPSRAG's agents are equipped with a suite of routing tools as outlined in Table 2.

5 DEPSRAG Proof-of-Concept

This section describes the implementation of DEPSRAG. It then describes the conducted experiments to evaluate DEPSRAG.

5.1 Implementation

We implemented DEPSRAG in Python using Langroid [11], which is a framework that supports the development of multi-agent LLMs and enables seamless integration with various LLMs. Moreover, Langroid can orchestrate the interaction between agents and contains built-in tools that facilitate the development of RAG applications, such as performing Web search and accessing Neo4j [4], a graph database with a declarative query language called Cypher. To address the orchestration and critic interaction requirements of DEPSRAG, we developed supplementary routing tools.

For instance, `QuestionTool` was designed to specifically relay decomposed questions from the `AssistantAgent` to the appropriate agent, as determined by the "target_agent" flag generated within the tool message (see Sub-Tasks 2 and 3 in Figure 1). While "ForwardTool" is a built-in tool in Langroid for routing messages between agents.

KG Schema. The user inputs the package name, version, and ecosystem, after which DEPSRAG generates the dependency graph based on the schema depicted in Figure 5. According to this schema, the nodes in the KG belong to the entity Package, which has only two properties, the package name and version. For entity relationships, there is only one relation "depends_on".

Source/Dataset of Software Dependencies. To obtain the dependencies for the provided project and construct the dependency KG, we used Deps.Dev API,[4] a service developed and hosted by Google to help developers understand the structure and security of open-source software packages. This service was used in previous work [28, 21] to characterize and analyze dependencies. It repeatedly examines websites, such as GitHub and PyPI, to find up-to-date information about open-source software packages, thus generating a comprehensive list of direct and transitive dependencies that are constantly updated. DEPSRAG receives a JSON response from the Deps.Dev API that contains a list of dependencies of the provided package. Then ConstructKGTool propagates the data in the JSON data to build the dependency KG.

5.2 Experiments

In this study, we address the following research questions (RQs):

- RQ1: Does `DependencyGraphAgent` generate syntactically and functionally accurate Cypher queries to retrieve information from the dependency KG?
- RQ2: Does Critic-Agent interaction, the core design pattern underlying DEPSRAG, effectively enhance the reliability of DEPSRAG?

We have chosen two representative LLMs to test DEPSRAG, summarized in Table 3. Both are pretrained models. GPT-4 Turbo represents a proprietary model, whereas Llama-3 is an open-source model.

5.2.1 RQ1: Accuracy of Cypher Query Generation

The effectiveness of the agent, `DependencyGraphAgent`, is contingent on the LLM's ability to accurately translate user queries from natural language into Cypher queries for retrieving information from Neo4j database, which stores the dependency KG. Consequently, it is essential to assess the accuracy of the generated Cypher queries from both syntactical and functional perspectives to ensure reliable performance.

To evaluate this aspect, we asked DEPSRAG to answer questions about important properties of graphs (see Table 1) based on the two models selected in this work (listed in Table 3). We conducted this experiment without involving the `CriticAgent`, since according to our implementation, `CriticAgent` provides only a feedback on the final answer generated by `AssistantAgent`. The questions presented in Table 1 are asked after constructing the dependency graph for `Chainlit` version 1.1.200.

We observed that both LLMs are capable of generating correct Cypher queries, though not consistently. To address this, we implemented a strategy where the LLMs first retrieve the database schema before generating the query and are provided with the error message to prompt a retry if necessary. As shown in Table 1 (column 3), Llama-3 failed to produce correct Cypher queries on the first attempt for two questions, requiring one retry for the first question and two for the second. In contrast, GPT-4-Turbo successfully generated accurate Cypher queries on the first attempt for all questions. The final column of Table 1 indicates whether the provided answer was correct (without utilizing the `CriticAgent`). Despite Llama-3 producing a syntactically correct Cypher query after two attempts, the resulting query yielded an inaccurate answer. This occurred because the generated query was overly general, returning an overall count of paths without explicitly considering the node

[4] https://deps.dev

Chainlit as the starting point for each path, leading to an irrelevant answer in the context of the dependency graph.

Table 1: The performance of DependencyGraphAgent to generate accurate Cypher queries based on the selected LLMs after constructing the dependency graph for Chainlit version 1.1.200. DEPSRAG is executed without involving the CriticAgent.

Question	# of Cypher Query Trials	Model	Correct Response
What is the depth of the graph?	0	GPT-4-Turbo	Yes
	1	Llama-3	Yes
Are there cycles in the graph?	0	GPT-4-Turbo	Yes
	0	Llama-3	Yes
How many path chains are in the graph?*	0	GPT-4-Turbo	Yes
	2	Llama-3	No

* Listing 1 presents the Cypher queries generated by both Llama-3 and GPT-4-Turbo.

5.2.2 RQ2: Efficiency of the Agent-Critic interaction

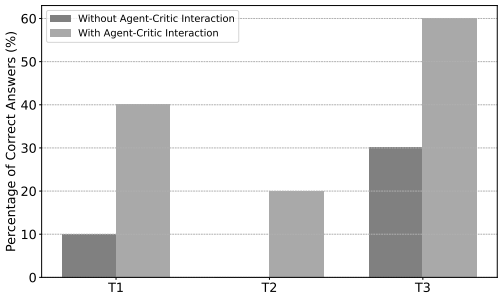

Figure 3: Ablation results show the correctness of the answers with and without Agent-Critic interaction.

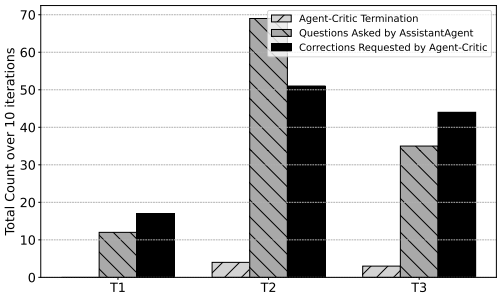

Figure 4: Total number of terminatiom, AssistantAgent questions, and CriticAgent responses.

We performed an ablation study to assess the effectiveness of the Agent-Critic interaction in DEPSRAG. Specifically, we compare the precision of the answers for three tasks (Listing 2) in ten iterations with and without CriticAgent. In these experiments, we focus exclusively on GPT-4-Turbo, as our preliminary observations indicated that Llama-3 encountered difficulties following the orchestration and Critic interaction outlined in Figure 2. We restricted the Agent-Critic interaction to the final answer generated by the AssistantAgent. This decision was made to minimize the number of iterative exchanges with other agents, as increasing these iterations would negatively impact performance by increasing token costs and runtime, while also reducing reliability [13, 8]. The results are presented in Figure 3. We manually verified the accuracy of the answers in both experiments and observed that the Agent-Critic interaction improves the quality and correctness of the answers. Specifically, the ablation results significantly improved accuracy with the Agent-Critic mechanism. Without CriticAgent, the precision was 13.3%, rising to 40% after its integration, highlighting the impact of CriticAgent on the accuracy of the response. Subsequently, introducing the agent-critic resulted in the DEPSRAG being **three times** more accurate.

We extend our analysis of the Agent-Critic interaction by examining the frequency of CriticAgent interventions to correct errors in the responses generated by AssistantAgent. Instances where CriticAgent provides feedback, are characterized by multiple rounds of interaction between the AssistantAgent and CriticAgent. The findings of this analysis, including the total number of terminations, are illustrated in the histogram in Figure 4. In certain cases, we observe unproductive exchange cycles between CriticAgent and AssistantAgent, where no progress is achieved. To address this, we impose a limit of ten feedback iterations per interaction, after which the session is terminated, and the final response is recorded. We also tracked the total number of questions posed by the AssistantAgent, as shown in Figure 4. The data reveal that $T2$ involved a higher number of questions, which is attributed to the AssistantAgent querying the vulnerability of each identified package individually. This increase in question count is quantified by monitoring the use of the QuestionTool.

6 Related Work

Critic and Feedback in LLM Apps. Konda et al. [20] presented the Actor-Critic Algorithm, which is a type of Reinforcement Learning (RL) algorithm that combines aspects of both policy-based methods (i.e., actor) and value-based methods (i.e., critic). This hybrid approach is designed to address the limitations of each method when used individually. In our approach, we adapt the CRITIC framework proposed by Gou et al. [17] to reinforce LLM agents not by updating weights, which is the case in RL, but through linguistic feedback from critic agents in a reflective, feedback loop. Alhanahnah et al. [8] present a multi-agent tool for repairing Alloy specifications, with feedback agents but lacking retrieval agents like `DependencyGraphAgent` and `SearchAgent`, which are used in DEPSRAG. MALADE [13], a multi-agent system for answering medication side-effect queries, employs Critic-Agent interaction and retrieves data from unstructured documents. DEPSRAG, by contrast, primarily uses a knowledge graph and the Web for retrieval.

Software representation as KG. Litzenberger et al. [22] proposed a unified data model to implement and construct dependency graphs for arbitrary repositories, thus facilitating the comparison of dependency graphs between different repositories. In their implementation, the data model is based on Neo4j, which makes is compliant with DEPSRAG's KG schema. Maninger et al. [24] proposed a visionary approach involving KGs to create a trustworthy AI software development assistant. Specifically, KGs can enable LLMs to correctly and appropriately explain the generated code. To our knowledge, we are the first to investigate the ability of LLMs to generate KGs for software dependencies, and utilize these graphs to aid in responding to queries concerning the dependency structure. Musco et al. [26] constructed dependency graphs for Java programs at the class level to understand commonalities between different Java projects. In contrast, DEPSRAG supports four software ecosystems and constructs dependency graphs at the package level.

7 Conclusion and Future Work

We presented DEPSRAG, the first steps of an agent-based approach for automated planning and reasoning about software dependencies. Our evaluation showed that Agent-Critic interaction enhances the quality, correctness, and reasoning of LLM responses. However, implementing this architecture necessitates efficient orchestration and mechanisms for preventing feedback loops. Future work will explore several key directions:

- **Evaluating different Agent-Critic Settings.** In this work, we designed and implemented `CriticAgent` to provide feedback on the final answer generated by `AssistantAgent`, which integrates responses from the retriever agents (`DependencyGraphAgent` and `SearchAgent`). However, it is crucial to assess various scenarios to optimize the effectiveness of Agent-Critic interactions.

- **Generating Software Bill of Materials (SBOM).** The generated dependency graph encompasses both direct and transitive dependencies, enhancing the suitability of DEPSRAG for SBOM generation and providing metadata such as path chains (that is, dependency hierarchy [9]), as opposed to presenting a flat list of dependencies. SBOM format specifications, such as CycloneDX [1], are designed to accommodate this form of structured data. The hierarchical structure of the dependencies upheld by DEPSRAG is in accordance with the minimum SBOM requirements prescribed by the US National Telecommunications and Information Administration (NTIA). DEPSRAG exceeds these regulatory requirements, which require disclosure of primary dependencies (first-level) and all ensuing transitive dependencies (second-level) [5], by documenting the complete chain of dependencies. The generation of SBOM will enable DEPSRAG to support vulnerability management, which requires designing a dedicated retrieval to access different sources of vulnerability databases.

- **Dependency Resolution.** Updating dependencies is cumbersome and can introduce incompatibilities that break the application [14]. DEPSRAG maintains knowledge about all dependencies within the application, thus qualifying it to provide recommendations that suggest non-vulnerable package versions and will not lead to compatibility issues.

Acknowledgments and Disclosure of Funding

We would like to thank Benoit Baudry for the helpful discussions and feedback. This material is based on work supported by the Office of Naval Research (ONR) under Contract N00014-24-1-2049. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of ONR.

References

[1] CycloneDX/specification. https://github.com/CycloneDX/specification. [Accessed 10-05-2024].

[2] EU Cyber Resilience Act — digital-strategy.ec.europa.eu. https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act. [Accessed 30-05-2024].

[3] Executive Order 14028, Improving the Nation's Cybersecurity — nist.gov. https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity. [Accessed 31-05-2024].

[4] Neo4j Graph Database — neo4j.com. https://neo4j.com/product/neo4j-graph-database/. [Accessed 06-05-2024].

[5] ntia.doc.gov. https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf. [Accessed 13-05-2024].

[6] Garima Agrawal. Aiseckg: Knowledge graph dataset for cybersecurity education. *AAAI-MAKE 2023: Challenges Requiring the Combination of Machine Learning 2023*, 2023.

[7] Garima Agrawal, Tharindu Kumarage, Zeyad Alghami, and Huan Liu. Can knowledge graphs reduce hallucinations in llms?: A survey. *arXiv preprint arXiv:2311.07914*, 2023.

[8] Mohannad Alhanahnah, Md Rashedul Hasan, and Hamid Bagheri. An empirical evaluation of pre-trained large language models for repairing declarative formal specifications, 2024.

[9] Musard Balliu, Benoit Baudry, Sofia Bobadilla, Mathias Ekstedt, Martin Monperrus, Javier Ron, Aman Sharma, Gabriel Skoglund, César Soto-Valero, and Martin Wittlinger. Challenges of producing software bill of materials for java. *IEEE Security & Privacy*, 2023.

[10] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. The maven dependency graph: a temporal graph-based representation of maven central. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, page 344–348. IEEE Press, 2019.

[11] Prasad Chalasani, Nils Palumbo, Mohannad Alhanahnah, and Somesh Jha. Langroid: Multi-agent framework for llm applications. https://github.com/langroid/langroid, 2023.

[12] Harrison Chase. LangChain, 2022. Accessed: 2024-02-27.

[13] Jihye Choi, Nils Palumbo, Prasad Chalasani, Matthew M Engelhard, Somesh Jha, Anivarya Kumar, and David Page. Malade: Orchestration of llm-powered agents with retrieval augmented generation for pharmacovigilance. *arXiv preprint arXiv:2408.01869*, 2024.

[14] Andreas Dann, Ben Hermann, and Eric Bodden. Upcy: Safely updating outdated dependencies. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 233–244. IEEE, 2023.

[15] Johannes Düsing and Ben Hermann. Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories. *Digital Threats*, 3(4), feb 2022.

[16] Raimar Falke, Raimund Klein, Rainer Koschke, and Jochen Quante. The dominance tree in visualizing software dependencies. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–6. IEEE, 2005.

[17] Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*, 2023.

[18] Yanchao Hao, Yuanzhe Zhang, Kang Liu, Shizhu He, Zhanyi Liu, Hua Wu, and Jun Zhao. An end-to-end model for question answering over knowledge base with cross-attention combining global knowledge. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 221–231, 2017.

[19] Carl Hewitt. Actor model of computation: Scalable robust information systems, 2015.

[20] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

[21] Elizabeth Lin, Igibek Koishybayev, Trevor Dunlap, William Enck, and Alexandros Kapravelos. Untrustide: Exploiting weaknesses in vs code extensions. 2024.

[22] Tobias Litzenberger, Johannes Düsing, and Ben Hermann. Dgmf: Fast generation of comparable, updatable dependency graphs for software repositories. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 115–119, 2023.

[23] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

[24] Daniel Maninger, Krishna Narasimhan, and Mira Mezini. Towards trustworthy ai software development assistance. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 112–116, 2024.

[25] Parastoo Mohagheghi and Reidar Conradi. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering*, 12:471–516, 2007.

[26] Vincenzo Musco, Martin Monperrus, and Philippe Preux. A generative model of software dependency graphs to better understand software evolution, 2017.

[27] Jeremy Rack and Cristian-Alexandru Staicu. Jack-in-the-box: An empirical study of javascript bundling on the web and its security implications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 3198–3212, 2023.

[28] Imranur Rahman, Nusrat Zahan, Stephen Magill, William Enck, and Laurie Williams. Characterizing dependency update practice of npm, pypi and cargo packages. *arXiv preprint arXiv:2403.17382*, 2024.

[29] Tidelift. 2022 open source supply chain survey — Tidelift — tidelift.com. https://tidelift.com/2022-open-source-software-supply-chain-survey. [Accessed 24-04-2024].

[30] Chenyan Xiong, Russell Power, and Jamie Callan. Explicit semantic ranking for academic search via knowledge graph embedding. In *Proceedings of the 26th international conference on world wide web*, pages 1271–1279, 2017.

[31] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE symposium on security and privacy*, pages 590–604. IEEE, 2014.

[32] Bishan Yang and Tom Mitchell. Leveraging knowledge bases in lstms for improving machine reading. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1436–1446, 2017.

[33] Fuzheng Zhang, Nicholas Jing Yuan, Defu Lian, Xing Xie, and Wei-Ying Ma. Collaborative knowledge base embedding for recommender systems. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 353–362, 2016.

A Appendix / supplemental material

Table 2: Summary of tools used by DEPSRAG Agents.

Tool Name	Corresponding Agent	Purpose
ConstructKGTool	`DependencyGraphAgent`	Construct the KG
GraphSchemaTool	`DependencyGraphAgent`	Obtain the KG schema
CypherQueryTool	`DependencyGraphAgent`	Receive the generated query, execute it, and return the result
VisualizeKGTool	`DependencyGraphAgent`	Visualize the KG
WebSearchTool	`SearchAgent`	Perform Web search and return the most relevant
VulnerabilityTool	`SearchAgent`	Search for vulnerabilities on OSV vulnerability database
UserInteractionTool	`AssistantAgent`	Control the user interaction with the agents
QuestionTool	All except `CriticAgent`	Inter-agent routing
ForwardTool	`AssistantAgent` and `CriticAgent`	Inter-agent routing

Table 3: Characteristics of the selected LLMs

Model	Version	Cut-off	Context Window (Tokens)	Input Cost per 1M tokens	Output Cost per 1M tokens
GPT-4 Turbo	1106-preview	Apr 2023	128k	$10	$30
Llama-3	70b-instruct	Dec 2023	8k	N/A	N/A

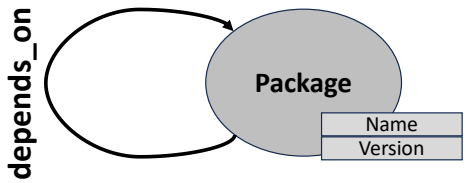

Figure 5: Dependency KG Schema consisting of a single entity with two attributes, “name” and “version”, and one relation.

Listing 1: Cypher queries generated by both models translating question 3 in Table 1

```
-- Query generated by Llama-3
MATCH p=()-[r:DEPENDS_ON*]->() RETURN count(p)
AS pathCount

-- Query generated by GPT-4-Turbo
MATCH p=(root:Package {name: 'chainlit', version: '1.1.200'})-[:
    DEPENDS_ON*]->(leaf:Package)
WHERE NOT (leaf)-[:DEPENDS_ON]->() RETURN count(p)
AS pathCount
```

Listing 2: List of tasks used in RQ3

```
{
    "T1": "what's the density of the dependency graph of
       chainlit version 1.1.200 pypi",

    "T2": "which packages in chainlit version 1.1.200 pypi have
        the most dependencies relying on them (i.e., nodes
       have the highest in-degree in the graph), and what is
       the risk associated with a vulnerability in those
       packages?",

```

```
    "T3": "In the dependency graph of chainlit version 1.1.200
       pypi, are there any multi-version conflicts where
       different packages depend on different versions of the
       same package? If yes, provide examples of these
       conflicts and all paths that lead to these packages
       from the root node"
}
```