LLM Agents for Automated Dependency Upgrades

Vali Tawosi
J.P. Morgan AI Research
London, UK
vali.tawosi@jpmorgan.com

Salwa Alamir
J.P. Morgan AI Research
London, UK
salwa.alamir@jpmchase.com

Xiaomo Liu
J.P. Morgan AI Research
New York, USA
xiaomo.liu@jpmchase.com

Manuela Veloso
J.P. Morgan AI Research
New York, USA
manuela.veloso@jpmchase.com

***Abstract*—As a codebase expands over time, its library dependencies can become outdated and require updates to maintain innovation and security. However, updating a library can introduce breaking changes in the code, necessitating significant developer time for maintenance. To address this, we introduce a framework of LLM agents to be used in combination with migration documentation to automatically recommend and apply code updates and ensure compatibility with new versions. Our solution can automatically localize updated library usages in live Java codebases and implement recommended fixes in a user-friendly manner. The system architecture consists of multiple key components: a Summary Agent, Control Agent, and Code Agent. To validate our approach, we apply the framework on an industrial use case by which we create three synthetic code repositories with major Upgrade changes and benchmark our approach against state-of-the-art methods. Results show that our approach not only performs upgrades using fewer tokens across all cases but also achieves a precision of 71.4%, highlighting its efficiency and effectiveness compared to state-of-the-art methods.**

***Index Terms*—AI for SE, Agent-Based SE, LLM for Library Upgrades**

I. Introduction

Modern software projects strive to avoid reinventing the wheel by leveraging reusable functionality provided by open-source libraries. However, updating these libraries can introduce breaking changes in the code, posing challenges for developers [1]. Many software projects heavily depend on open-source libraries that offer ready-to-use functionality, including those available for Java. For example, Maven Central is a repository for Java libraries, providing developers with access to a vast array of libraries for their projects [2]. While these libraries offer convenience and accessibility, they also introduce dependencies, and frequent library updates can result in breaks in developers' code.

In the Java ecosystem, contributors often use warnings to alert library users of upcoming changes to the API and functionality. These warnings are typically issued several releases before the actual changes take effect, allowing developers time to update their code for compatibility [3]. However, there is currently no clearly established method for automatically updating library usages in code to account for changes in Java libraries, leading to costly and time-consuming manual maintenance without immediate incentives. This issue is particularly pronounced in larger codebases, where developers might ignore warnings or lock into a specific library version [4], [5]. Such strategies can hinder a team's ability to innovate in the long term, preventing them from benefiting from new features, performance improvements, and bug fixes offered by updated libraries.

Some automated solutions have been proposed in the past, but have not been widely adopted, giving way to further research opportunities in this domain. Furthermore, large language models (LLMs) have been at the forefront of AI, and the software engineering domain is no exception. LLM systems have been used to tackle a number of coding challenges, with a focus on code generation and code completion [6], or automated program repair [7]. Nevertheless, the maintenance and evolution of code remains a critical task in engineering, impacting code quality and reusability [8]. As such, in this paper we propose a framework of LLM Agents for automated dependency upgrades. This system builds upon our previous work, the Autonomous LLM-based Multi-Agent Software Engineering Framework (ALMAS) [9]. The system is evaluated on an industrial use case with the aim to update code repositories after completing a major library upgrade.

II. Related Work

Automated codebase updates for dependency compatibility have been explored previously, often by analyzing differences between library versions. SemDiff [10] recommends client code adaptations by tracking framework changes, while LIBSYNC [11] learns API adaptation patterns from clients that have already migrated to newer versions.

Most prior work focuses on statically typed languages, though recent efforts address dynamic languages. For example, Nielsen et al. use semantic patches to locate breaking changes in JavaScript [12]. Other approaches include manual changelog labeling, basic NLP for error-documentation matching [13], and neural methods.

Neural machine translation (NMT) models, adapted from NLP, treat code tasks like bug fixing [14], summarization [15], and review automation [16] as sequence-to-sequence problems using LSTM [17] [18] or Transformer architectures [19]. However, source code generation faces challenges in syntax and semantics, prompting solutions like syntax-aware decoders [20] and AST-based methods [21]–[23].

Recent work leverages pre-trained code language models (CodeLMs) such as CodeBERT, CodeGPT, and CodeT5 for code updates. Liu et al. found that CodeLMs struggle with time-sensitive, complex updates and generalization [24]. PCREQ, a six-component system, outperforms popular LLMs

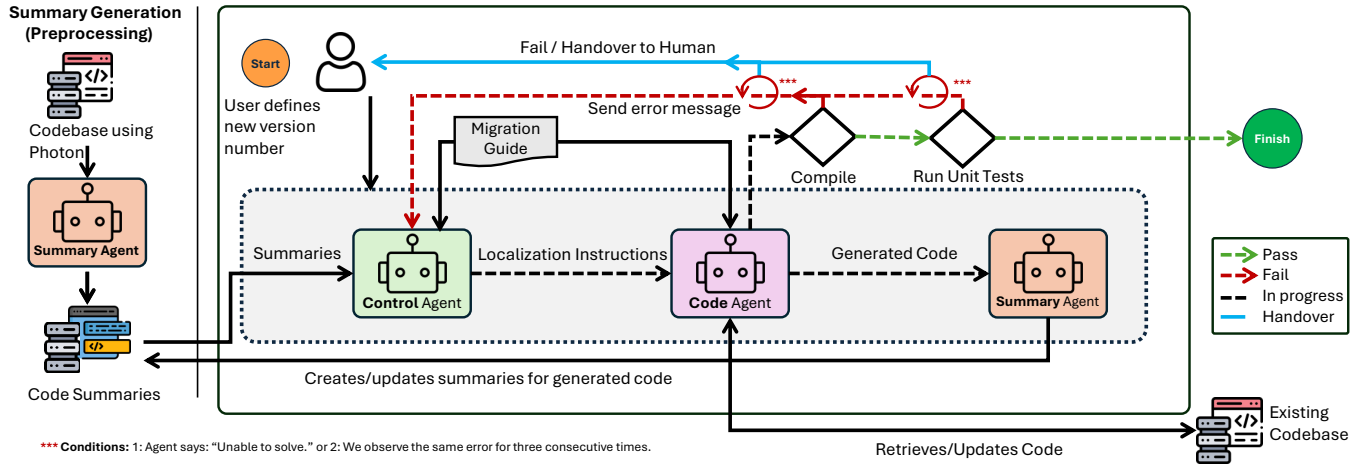

Fig. 1: Diagram of the multi-agent LLM framework for automated Java dependency upgrades, showing the process of code summarization, change localization, and implementation, with iterative compilation and testing, and handover to a human.

by 18–20% [25]. Multi-agent LLM systems have also shown promise in automated program repair on benchmarks like SWE-Bench [26].

Our work frames code updates as sequential bug-fixing tasks, using a multi-agent LLM system [9] that utilises code summaries, iteratively updates code, and ensures successful builds and tests, with automatic handover to human intervention when needed.

III. Methodology

Our approach (depicted in Fig. 1) leverages Meta-RAG, a change localisation mechanism designed for efficient code editing in large codebases by summarising code into structured natural language, reducing token count by nearly 80% and enabling scalable information retrieval for localisation tasks [27]. This summarisation step provides a concise, coherent overview of the codebase, facilitating downstream editing and upgrades. The main workflow is built on a customised ALMAS multi-agent framework, which iteratively applies changes and resolves issues until the project builds and all tests pass, mirroring standard developer upgrade practices [9].

A. Preprocessing

As a preprocessing step, the *Summary Agent* generates and stores code summaries aligned with the code's Abstract Syntax Tree (AST). Each code file and its units (functions/classes) receive one-line summaries describing their responsibilities, ensuring accurate mapping between code and summary structures [27]. These summaries are then used for retrieval and localisation, allowing RAG to operate on metadata rather than raw code. Summarisation is performed once for the entire codebase, with incremental updates as the code evolves.

B. Main process

The main process (depicted in Fig. 1 inside the gray box) makes use of a number of agents and knowledge sources to complete the upgrade. We describe the agents and their responsibilities before diving into the process workflow.

1) Control Agent: The Control Agent, central to Meta-RAG and ALMAS [9], identifies code units to read (for context) and to write (for modification or creation) based on the current task. It first examines file-level summaries to select relevant files, then drills down to finer-grained summaries to pinpoint code units for context and editing. This enables efficient code retrieval and encourages code reuse and natural integration of changes.

2) Code Agent: The Code Agent (e.g., GPT-4o) receives the task and selected code units, then implements the required changes. By minimising context length, the agent avoids repeated code searches typical of other approaches. After editing, it triggers the Summary Agent to update summaries, maintaining alignment between code and metadata

3) Process Workflow: The upgrade process starts with user initiation, specifying a target version or retrieving it automatically from artifactory. The control agent accesses the codebase and generates a structure for every pom and yml file in the project, as these files are not included in the summary base. The Control Agent also consults migration guides if available. It generates upgrade instructions, identifying files and changes needed. The Code Agent applies these changes, updating dependencies and configurations. The project is then compiled and tests ran; if errors occur, logs are sent back to the Control Agent for further resolution, following an Automated Program Repair (APR) loop. The cycle of applying changes, compiling the code, running tests, and reviewing the output log continues until the build and test run is successful. The code agent indicates its inability to resolve the issue or if the same error occurs more than n times (with $n = 3$), the run stops. These conditions are implemented to prevent an infinite loop that could exhaust resources. The framework supports collaboration, allowing handover to human developers with a summary of actions taken (stored while the process was

TABLE I: Comparative results of the multi-agent LLM framework (LADU) against benchmark, showing the number of files, common files with Gold (G), lines added or removed, and common lines with (G) for each version upgrade scenario.

Approach	Start	Target	#Files	#File (G)	#Common Files	#Lines A:R	#Lines A:R (G)	#Common Lines
OpenHands + Claude 3.7 Sonnet	3.1.12	3.2.11	8	7	7	33:78	38:62	14:45
OpenHands + Claude 3.7 Sonnet	3.2.11	3.3.9	7	9	5	2:128	50:37	2:22
OpenHands + Claude 3.7 Sonnet	3.3.9	3.4.2	7	7	6	6:132	23:31	4:9
LADU + GPT-4o	3.1.12	3.2.11	8	7	7	37:67	38:62	13:48
LADU + GPT-4o	3.2.11	3.3.9	3	9	3	4:7	50:37	2:5
LADU + GPT-4o	3.3.9	3.4.2	3	7	3	10:9	23:31	4:5

TABLE II: Execution results of the multi-agent LLM framework (LADU) against benchmark.

Metric	OpenHands			LADU		
	3.1	3.2	3.3	3.1	3.2	3.3
Number of steps	106	86	72	18	16	4
Execution runtime [s]	1020.0	720.0	450.0	473.6	424.0	277.4
Total Tokens	1,514,456	822,781	594,366	78,421	54,812	14,387
Cost [$]	4.66	2.38	1.83	0.62	0.45	0.11

running), and resumption by AI agents if needed.

IV. Upgrades Use Case

We assess our framework by upgrading a Java Moneta-based system across three synthetic repositories. This section outlines the repositories, explains the rationale for using synthetic data, and presents results comparing our framework's upgrades to manual developer upgrades ("Gold standard") and to the OpenHands agentic development tool [28].

A. Synthetic Dataset

The target dependency is a cloud-native, event-sourced microservice framework for Java, built on Spring Boot and widely used in industry. Upgrades often require multiple code changes across repositories, but not all updates are consistently applied. To ensure comprehensive evaluation, we created three synthetic repositories that encompass all necessary upgrade changes. These repositories were manually upgraded to serve as the Gold standard for comparison. They reflect real-world industrial setups, including Java source files, yml configurations, and pom files.

B. Results

The framework is evaluated using the aforementioned three synthetic code repositories against the "Gold" (G) standard, each representing different upgrade scenarios for the framework. The repositories are designed to test the framework's ability to apply all necessary updates for specific version transitions: from versions 3.1 to 3.2, 3.2 to 3.3, and 3.3 to 3.4. We select a baseline (OpenHands) to benchmark against our **L**LM **A**gents for **D**ependency **U**pgrades (LADU).

Table I compares the framework's performance with OpenHands + Claude 3.7 Sonnet and LADU + GPT-4o approaches. Key metrics include the number of files updated by each approach, the number of files in common with (G), the number of lines added or removed (A:R) by each approach, and common lines between the framework's output and (G). The common file and line metrics are calculated as an exact match between the patch diffs. This is because LADU is run until unit tests pass, but the tests may not be comprehensive and do not cover deployment-related configurations in yaml files.

Table II presents a comparative analysis of the multi-agent LLM framework (LADU) against the OpenHands benchmark with regards to the execution metrics in order to assess efficiency. Number of steps, execution runtime in seconds, total tokens and cost in USD are reported.

C. Discussion

The results indicate that the proposed framework, while not perfectly matching the "gold" standard, demonstrates a significant ability to automate dependency upgrades with a reasonable degree of accuracy. The number of common files and lines suggests that the framework can effectively identify and implement necessary changes, although there is room for improvement in aligning with manual upgrades.

The first version upgrade was presented with the most comprehensive migration guide, which is reflected in the performance of both approaches in Table I. LADU is able to achieve comparable performance to the baseline whilst using GPT-4o and comprising of a much simpler codebase and structure. More specifically, with regards to recall, our framework is able to make more correct line changes with regards to the addition of relevant lines, but does not remove as many correct ones as OpenHands.

In terms of precision, LADU attempts fewer changes, resulting in higher precision and reducing the risk of unwanted modifications. For example, in the 3.2 to 3.3 upgrade, LADU's removals were more accurate (71.4% vs. OpenHands' 17.2%).

Table II shows LADU requires far fewer steps (18 vs. 106 for OpenHands) and runs faster. It also uses significantly fewer tokens, especially in version 3.3, thanks to Meta-RAG's codebase condensation. LADU's token count includes summary generation, yet it still achieves results comparable to state-of-the-art benchmarks with greater efficiency.

V. Conclusion

This paper presents a novel framework consisting of multiple agents, including a Summary Agent, Control Agent, and Code Agent, which work together to update outdated library usages in codebases, while ensuring compatibility with new versions. The framework employs a change localization mechanism (Meta-RAG), which condenses the codebase, facilitating efficient information retrieval. The system was evaluated using

synthetic code repositories, demonstrating its ability to automate upgrades with precision and efficiency compared to existing solutions. In the future, the evaluation can be expanded to include real-world codebases, and robust unit tests, to provide a more comprehensive assessment of the framework's effectiveness in industrial settings. Furthermore, integrating more advanced LLMs and exploring hybrid approaches might also improve performance. Overall, the framework represents a promising advancement in automated software maintenance, offering a scalable solution for managing Java dependency upgrades in large codebases.

DISCLAIMER

REFERENCES

[1] M. Keshani, S. Vos, and S. Proksch, "On the relation of method popularity to breaking changes in the maven ecosystem," *Journal of Systems and Software*, vol. 203, p. 111738, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223001334

[2] N. Ede, J. Dietrich, and U. Zülicke, "Popularity and innovation in maven central," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, Apr. 2025, p. 334–338. [Online]. Available: http://dx.doi.org/10.1109/MSR66628.2025.00061

[3] Z. Zhong, S. He, H. Wang, B. Yu, H. Yang, and P. He, "An empirical study on package-level deprecation in python ecosystem," 2024. [Online]. Available: https://arxiv.org/abs/2408.10327

[4] A. Sawant, R. Robbes, and A. Bacchelli, "To react, or not to react: Patterns of reaction to api deprecation," *Empirical Software Engineering*, vol. 24, 12 2019.

[5] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration," *Empirical Software Engineering*, vol. 23, no. 1, 2017. [Online]. Available: http://dx.doi.org/10.1007/s10664-017-9521-5

[6] M. C. et al., "Evaluating large language models trained on code," 2021.

[7] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" *CoRR*, vol. abs/2310.06770, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.06770

[8] S. Raemaekers, A. Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the maven repository," *Journal of Systems and Software*, vol. 129, 04 2016.

[9] V. Tawosi, K. Ramani, S. Alamir, and L. Xiaomo, "ALMAS: an autonomous LLM-based multi-agent software engineering framework," in *2025 IEEE/ACM International Workshop on on Multi-Agent Systems using Generative Artificial INtelligence for Automated Software Engineering (MAS-GAIN)*. IEEE, 2025.

[10] B. Dagenais and M. P. Robillard, "Semdiff: Analysis and recommendation support for api evolution," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 599–602.

[11] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 302–321, 2010.

[12] B. B. Nielsen, M. T. Torp, and A. Møller, "Semantic patches for adaptation of javascript programs to evolving libraries," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 74–85.

[13] A. Ni, D. Ramos, A. Z. H. Yang, I. Lynce, V. Manquinho, R. Martins, and C. Le Goues, "Replication of soar: A synthesis approach for data science api refactoring," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2021, pp. 190–191.

[14] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1161–1173, 2021.

[15] A. Mastropaolo, S. Scalabrino, N. Cooper, D. Nader, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," 05 2021, pp. 336–347.

[16] V. J. Hellendoorn, J. Tsay, M. Mukherjee, and M. Hirzel, "Towards automating code review at scale," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1479–1482. [Online]. Available: https://doi.org/10.1145/3468264.3473134

[17] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, ser. AAAI'17. AAAI Press, 2017, p. 1345–1351.

[18] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Martin, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, pp. 1943–1959, 2021.

[19] N. Chirkova and S. Troshin, "Empirical study of transformers for source code," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 703–715. [Online]. Available: https://doi.org/10.1145/3468264.3468611

[20] M. Zhu, Y. Zhang, W. Chen, M. Zhang, and J. Zhu, "Fast and accurate shift-reduce constituent parsing," in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Sofia, Bulgaria: Association for Computational Linguistics, Aug. 2013, pp. 434–443. [Online]. Available: https://aclanthology.org/P13-1043

[21] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 150–162.

[22] S. Alamir, P. Babkin, N. Navarro, and S. Shah, "Ai for automated code updates," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022, pp. 25–26.

[23] N. Navarro, S. Alamir, P. Babkin, and S. Shah, "An automated code update tool for python packages," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2023, pp. 536–540.

[24] Y. Liu, C. Tantithamthavorn, Y. Liu, P. Thongtanunam, and L. Li, "Automatically recommend code updates: Are we there yet?" 2024. [Online]. Available: https://arxiv.org/abs/2209.07048

[25] H. Lei, G. Xiao, Y. Liu, and Z. Zheng, "Pcreq: Automated inference of compatible requirements for python third-party library upgrades," 2025. [Online]. Available: https://arxiv.org/abs/2508.02023

[26] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," *arXiv preprint arXiv:2405.15793*, 2024.

[27] V. Tawosi, S. Alamir, X. Liu, and M. Veloso, "Meta-rag on large codebases using code summarization," *arXiv preprint arXiv:2508.02611*, 2025.

[28] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh *et al.*, "Openhands: An open platform for ai software developers as generalist agents," *arXiv preprint arXiv:2407.16741*, 2024.