European International Journal of Multidisciplinary Research and Management Studies

TYPE Original Research
PAGE NO. 194-202

Check for updates

Artificial Intelligence-Assisted Refactoring and Architectural Transformation: A Comprehensive Study on Large Language Models for Migrating Monolithic Software Systems to Microservices

OPEN ACCESS

SUBMITTED 16 September 2025
ACCEPTED 05 October 2025
PUBLISHED 31 October 2025
VOLUME Vol.05 Issue10 2025

Kim Keller

Department of Computer Science University of Zurich, Switzerland

Abstract: The rapid evolution of software systems and the increasing complexity of enterprise-scale applications have intensified the need for architectural modernization. Traditional monolithic software architectures, while historically dominant due to their simplicity and unified deployment model, have become increasingly difficult to maintain, scale, and evolve in contemporary distributed computing environments. Consequently, microservices architecture has emerged as a widely adopted paradigm for building scalable, modular, and independently deployable systems. However, the migration from monolithic architectures to microservices presents significant technical and organizational challenges, including service decomposition, dependency identification, data migration, and architectural restructuring. In recent years, the emergence of large language models (LLMs) and AI-assisted development tools has introduced new possibilities for automating aspects of software engineering, including code generation, code analysis, documentation, and architectural refactoring. This research article investigates the role of LLMs in facilitating the transformation of monolithic systems into microservices-based architectures. Drawing upon a comprehensive set of scholarly references on large language models, AI-assisted programming, prompt engineering, software refactoring strategies, and microservices extraction techniques, the study provides

a detailed theoretical and methodological exploration of AI-augmented architectural transformation. The article examines how LLM-driven tools can assist developers in identifying service boundaries, summarizing code structures, generating refactoring suggestions, and supporting iterative system decomposition. Additionally, the study analyzes the challenges associated with AI-assisted development, including hallucination, non-determinism in code generation, prompt sensitivity, and security vulnerabilities such as prompt-based exploitation. Through a conceptual methodological framework and interpretive analysis of existing research findings, the article evaluates both the capabilities and limitations of LLM-based software engineering assistance. The discussion further explores implications for future research, particularly regarding the integration of AI agents into automated software modernization pipelines. Ultimately, the article argues that while LLMs do not fully replace traditional software engineering practices, they represent a transformative tool that can significantly enhance developer productivity and accelerate architectural modernization when used responsibly within structured engineering workflows.

Keywords: Large Language Models, Microservices Architecture, Software Refactoring, AI-Assisted Programming, Monolithic Systems, Software Modernization, Prompt Engineering.

Introduction: Modern software systems have undergone a profound transformation in response to the increasing demands of scalability, reliability, and continuous delivery. In earlier stages of software engineering development, monolithic architectures represented the dominant approach for building enterprise applications. A monolithic system integrates all functional components of a software application into a single, unified codebase that is deployed as one cohesive unit. This architectural approach offered advantages in simplicity and ease of development, particularly during periods when applications were relatively small and computational environments were less distributed. However, as software systems expanded in complexity and scale, the limitations of monolithic architectures became increasingly evident.

One of the most significant challenges associated with monolithic systems is their limited flexibility in supporting independent component evolution. Because all application modules are tightly coupled within a single codebase, even minor changes may require recompilation and redeployment of the entire system. This constraint complicates maintenance processes, increases deployment risks, and slows development cycles. Furthermore, the tight coupling between modules often results in hidden dependencies that make it difficult to isolate functionality or understand system behavior (Fritzsch, Bogner, Zimmermann, & Wagner, 2018).

In response to these challenges, the microservices architectural paradigm has gained widespread recognition within both academic research and industrial practice. Microservices architecture decomposes large software systems into smaller, independently deployable services that communicate through lightweight protocols. Each microservice encapsulates a specific business capability and can be developed, deployed, and scaled independently from other services. This modular design supports continuous integration and deployment pipelines while enabling organizations to scale specific services according to workload demands (Knoche & Hasselbring, 2018).

Despite the clear benefits of microservices architecture, migrating from a monolithic system to a microservices-based architecture remains an extremely complex engineering task. The migration process typically involves identifying appropriate service boundaries, extracting functional modules from the monolith, refactoring code dependencies, and restructuring system data models. These tasks require extensive domain knowledge and careful analysis of the system's internal structure. Researchers have proposed several methodologies for identifying microservice candidates, including dataflow-driven analysis, dependency clustering, and service decomposition frameworks (Li et al., 2019; Gysel et al., 2016).

Recent advancements in artificial intelligence, particularly in the development of large language models (LLMs), have introduced new possibilities for automating aspects of software engineering tasks. LLMs are neural network models trained on vast corpora of textual and programming data, enabling them to generate natural language explanations, produce source code, and assist with software documentation. According to Zhao et al. (2023), large language models have demonstrated remarkable capabilities across multiple programming-related tasks, including code synthesis, code completion, debugging, and documentation generation.

The integration of LLMs into software development workflows has already begun to reshape programming practices. Tools such as AI-powered coding assistants allow developers to generate code snippets, receive automated suggestions, and explore alternative

implementation strategies. Experimental studies have shown that these systems can significantly improve programming productivity by assisting developers with repetitive or complex coding tasks (Poldrack, Lu, & Beguš, 2023). Similarly, research into prompt engineering has demonstrated that carefully structured natural language instructions can guide LLMs to generate useful solutions for programming problems (Denny, Kumar, & Giacaman, 2022).

Beyond simple code generation, researchers have begun exploring the potential of LLMs as autonomous software agents capable of performing multi-step reasoning tasks in programming environments. Such AI-based agents can analyze codebases, generate documentation, identify architectural patterns, and collaborate with developers during the software development lifecycle (Xi et al., 2023). These emerging capabilities suggest that LLMs could play a significant role in complex software engineering processes such as architectural refactoring and system modernization.

However, the adoption of LLM-based tools also introduces several critical challenges. One major concern is the phenomenon of hallucination, where language models generate plausible but incorrect information. In the context of software engineering, hallucinated code or documentation could introduce errors into production systems if not carefully validated by developers (Alkaissi & Samy, 2023). Additionally, research has shown that LLM outputs can be highly sensitive to prompt variations and may exhibit non-deterministic behavior when generating code solutions (Ouyang, Zhang, Harman, & Wang, 2023).

Security concerns have also emerged in relation to prompt manipulation and adversarial interactions with language models. Studies have demonstrated that carefully crafted prompts can bypass safety mechanisms or manipulate model outputs, potentially exposing vulnerabilities within AI-assisted development systems (Liu et al., 2023).

Despite these challenges, the potential for LLMs to support architectural modernization remains significant. AI-driven tools could assist developers in analyzing large codebases, generating summaries of system components, identifying potential service boundaries, and recommending refactoring strategies. When combined with established microservices migration techniques such as dataflow analysis and service decomposition frameworks, LLM-based systems may provide powerful assistance for transforming legacy software systems.

This research article investigates the intersection between large language models and architectural refactoring in the context of monolithic-to-microservices migration. By synthesizing insights from research on software architecture, AI-assisted programming, and microservices extraction methodologies, the article aims to provide a comprehensive theoretical framework for understanding how LLMs can support software modernization processes.

The study addresses several key research objectives. First, it examines the architectural challenges associated with monolithic systems and the motivations for adopting microservices architecture. Second, it explores existing methodologies for identifying microservices within legacy systems. Third, it analyzes the capabilities of large language models in software engineering tasks relevant to architectural transformation. Finally, it proposes a conceptual framework for integrating AI-assisted tools into software modernization workflows.

Through this extensive analysis, the article seeks to contribute to the growing body of literature on AI-driven software engineering while offering practical insights for organizations seeking to modernize their legacy systems. The findings suggest that while LLMs are not a complete solution for architectural refactoring, they represent a powerful complementary technology capable of enhancing human expertise and accelerating complex software engineering processes.

METHODOLOGY

The methodological framework adopted in this research is based on a qualitative analytical approach that synthesizes theoretical insights from multiple streams of existing academic literature. Rather than relying on a single experimental dataset or controlled laboratory experiment, the study integrates conceptual analysis, comparative evaluation, and interpretive synthesis of published research concerning large language models, AI-assisted software development, and microservices architecture migration strategies. This approach is particularly appropriate for emerging interdisciplinary domains where technological developments evolve rapidly and empirical evidence is distributed across multiple research areas.

The primary methodological objective is to construct a conceptual framework that explains how large language models can support the process of transforming monolithic software systems into microservices-based architectures. To accomplish this objective, the methodology is structured around three interconnected analytical stages: literature synthesis, capability mapping, and architectural workflow modeling.

The first stage of the methodology involves an extensive literature synthesis of the scholarly sources listed in the reference dataset. These references span several major

research domains including large language model architectures, AI-assisted programming tools, prompt engineering methodologies, code generation experiments, and software architectural refactoring strategies. By systematically examining these sources, the study identifies key theoretical concepts and technological capabilities relevant to software modernization.

Research on large language models provides foundational insights into the computational mechanisms that enable these systems to generate and interpret programming code. Large language models are typically based on transformer architectures trained on vast corpora of textual and programming data. Their ability to process both natural language and structured programming languages allows them to act as intermediaries between human developers and complex codebases (Zhao et al., 2023). These models can produce explanations of program logic, generate code fragments, summarize software components, and assist with debugging tasks.

Studies investigating AI-assisted programming provide empirical evidence regarding the practical capabilities of these systems. Experiments involving AI-assisted coding environments have demonstrated that developers can use language models to generate functional code snippets, propose alternative implementations, and explore architectural design patterns (Poldrack, Lu, & Beguš, 2023). Additionally, prompt engineering research has shown that carefully structured instructions significantly influence the quality and accuracy of generated programming solutions (Denny, Kumar, & Giacaman, 2022).

The second stage of the methodology focuses on capability mapping. In this phase, the technical capabilities identified in the literature are mapped onto the specific challenges associated with monolithic-to-microservices migration. Migrating a monolithic system typically involves several complex tasks including identifying service boundaries, analyzing code dependencies, extracting modular functionality, restructuring data models, and implementing inter-service communication mechanisms.

Previous research on microservices extraction techniques has proposed multiple strategies for performing service decomposition. Dataflow-driven approaches analyze the interactions between system components to identify clusters of related functionality that can form independent services (Li et al., 2019). Similarly, service decomposition frameworks such as Service Cutter utilize domain-driven design principles and system dependency analysis to define service boundaries (Gysel et al., 2016).

The methodology examines how large language models could assist with each stage of this process. For example, LLMs may help developers understand complex codebases by generating natural language summaries of modules and functions. Automatic code summarization capabilities have been explored in recent research, demonstrating that language models can provide concise descriptions of programming logic and structural relationships within code (Sun et al., 2023).

The third stage of the methodology involves constructing an architectural workflow model that integrates AI-assisted tools into the software modernization pipeline. This conceptual model outlines how human developers and AI systems can collaborate during different phases of the migration process.

The workflow begins with system exploration, during which developers analyze the existing monolithic codebase. At this stage, LLM-based tools may generate documentation and summaries of software modules, enabling engineers to understand legacy systems more efficiently. Because many enterprise monolithic systems lack comprehensive documentation, this capability represents a valuable productivity enhancement.

The next stage involves service candidate identification. Using insights from architectural decomposition methodologies, developers attempt to identify potential microservices based on functional boundaries and data dependencies. AI-assisted tools may support this stage by analyzing code segments and proposing clusters of related functionality.

Following service identification, developers perform code extraction and refactoring tasks to isolate each service from the monolithic system. During this stage, AI systems may assist with generating refactoring suggestions, identifying potential dependency conflicts, and producing alternative code implementations.

The final stage of the workflow involves validation and refinement. Because AI-generated outputs may contain inaccuracies, human oversight remains essential. Developers must carefully review generated code, validate system behavior, and ensure that refactored components satisfy architectural requirements.

In addition to these methodological steps, the study incorporates an analysis of known limitations associated with AI-assisted programming. Research has demonstrated that language models can exhibit non-deterministic behavior when generating code, meaning that identical prompts may produce different outputs across multiple executions (Ouyang, Zhang, Harman, &

Wang, 2023). This characteristic introduces uncertainty into automated development processes.

Security concerns also represent an important methodological consideration. Studies investigating prompt-based attacks have shown that malicious inputs may manipulate language model outputs or bypass safety restrictions (Liu et al., 2023). As a result, the integration of AI tools into software engineering workflows must incorporate safeguards to prevent exploitation.

By combining these analytical components, the methodology provides a comprehensive framework for examining how large language models may support architectural modernization. The approach emphasizes collaborative interaction between human developers and AI systems rather than complete automation. Such a hybrid model reflects the current technological reality in which AI tools enhance developer productivity while still requiring expert supervision and validation.

RESULTS

The interpretive analysis conducted in this study reveals several significant findings regarding the potential role of large language models in supporting the transformation of monolithic software systems into microservices architectures. These findings emerge from the synthesis of existing research across multiple domains including AI-assisted programming, software architecture modernization, and automated code analysis.

One of the most prominent findings concerns the capability of large language models to assist developers in understanding complex legacy systems. Monolithic enterprise software systems often evolve over many years or even decades, accumulating extensive codebases that contain numerous dependencies, undocumented modules, and outdated design patterns. In such environments, developers tasked with modernization efforts frequently struggle to comprehend the structure and functionality of the existing system.

Large language models demonstrate strong capabilities in generating natural language explanations of programming code. Through their training on large datasets containing programming languages and documentation, these models can interpret code segments and produce descriptive summaries that explain the purpose and functionality of specific components. Research on automatic code summarization indicates that language models can effectively translate programming constructs into human-readable descriptions, thereby supporting developer comprehension of unfamiliar codebases (Sun et al., 2023).

This capability has important implications for microservices migration projects. When engineers attempt to identify candidate services within a monolithic system, they must first understand the functional responsibilities of different modules. AI-generated summaries can significantly accelerate this exploration process by highlighting key operations, dependencies, and architectural relationships.

Another key finding relates to the potential use of LLMs in identifying service boundaries. Traditional microservices extraction techniques rely on analyzing system dependencies, data flows, and domain concepts to determine how a monolithic application can be decomposed into smaller services. Dataflow-driven methodologies have demonstrated effectiveness in clustering related functionalities based on interactions between system components (Li et al., 2019).

Large language models may complement these analytical techniques by interpreting code semantics and providing conceptual interpretations of system structure. For example, when provided with source code from a specific module, an LLM may infer the module's functional domain and suggest whether it represents a distinct business capability. Such insights could assist developers in evaluating whether a module should be extracted as an independent microservice.

Experimental research on AI-assisted coding environments provides additional evidence of the usefulness of language models in programming contexts. Studies have shown that developers can collaborate with AI systems to iteratively refine code implementations through conversational interaction. This process allows developers to explore multiple implementation strategies while receiving automated suggestions and explanations (Poldrack, Lu, & Beguš, 2023).

The results also indicate that LLM-based agents may support collaborative programming workflows. Recent research suggests that language models can function as autonomous agents capable of performing multi-step reasoning tasks, interacting with development tools, and coordinating multiple subtasks within a programming environment (Xi et al., 2023). Such capabilities could enable AI systems to assist with more complex refactoring operations that involve multiple stages of analysis and modification.

However, the findings also highlight several important limitations associated with AI-assisted software engineering. One significant concern is the phenomenon of hallucination. Language models occasionally generate information that appears plausible but is factually incorrect or logically

inconsistent. In the context of software engineering, hallucinated code segments or architectural recommendations could introduce errors if developers rely on AI-generated outputs without careful validation (Alkaissi & Samy, 2023).

Another limitation involves the non-deterministic behavior of language models during code generation tasks. Empirical studies have shown that identical prompts can produce different code outputs depending on factors such as temperature settings and internal probabilistic sampling processes. This variability can complicate efforts to reproduce consistent results within automated development pipelines (Ouyang, Zhang, Harman, & Wang, 2023).

Security vulnerabilities also represent a critical concern. Research investigating prompt engineering attacks has demonstrated that adversarial prompts may manipulate language model behavior in ways that bypass safeguards or produce unintended outputs. These vulnerabilities could pose risks in environments where AI tools are integrated directly into development infrastructure (Liu et al., 2023).

Despite these challenges, the overall results suggest that large language models can provide valuable assistance during software modernization projects. Rather than replacing human developers, these systems function most effectively as collaborative tools that augment human expertise.

The synthesis of findings indicates that AI-assisted development tools may significantly reduce the cognitive burden associated with analyzing large legacy systems. By generating documentation, summarizing code structures, and proposing refactoring suggestions, language models can help developers navigate complex codebases more efficiently.

Moreover, the integration of AI-based agents into development workflows may enable new forms of collaborative problem solving. Developers could interact with AI systems through conversational interfaces, iteratively refining architectural decisions and exploring alternative design strategies.

Taken together, the results highlight both the transformative potential and the practical limitations of large language models in the context of software architectural modernization. The next section examines these findings in greater depth through a comprehensive discussion of theoretical implications, practical considerations, and future research directions.

DISCUSSION

The findings presented in this study illuminate the emerging intersection between artificial intelligence technologies and software architectural modernization. The growing capabilities of large language models suggest that AI systems may become integral components of future software engineering workflows. However, the implications of these technologies extend beyond simple productivity improvements and raise deeper questions about the evolving relationship between human developers and intelligent computational tools.

One of the most significant theoretical implications concerns the role of artificial intelligence in augmenting human cognitive processes during software development. Traditional programming environments rely heavily on human expertise to interpret complex codebases, identify architectural patterns, and implement structural transformations. These tasks require extensive experience and often involve significant cognitive effort when dealing with large legacy systems.

Large language models introduce a new paradigm in which computational systems assist developers by performing preliminary analysis, generating explanations, and proposing solutions. In this sense, LLMs function as cognitive collaborators rather than purely mechanical tools. Their ability to translate code into natural language explanations bridges the gap between machine-readable instructions and human conceptual understanding.

This capability is particularly valuable in the context of legacy software modernization. Many monolithic enterprise systems were developed decades ago using programming practices that differ significantly from contemporary architectural paradigms. As a result, the engineers responsible for modernizing these systems often lack direct knowledge of the original design intentions. AI-generated summaries can partially reconstruct these intentions by analyzing code patterns and inferring functional responsibilities.

Another important implication concerns the evolution of software development methodologies. The emergence of AI-assisted programming tools may gradually shift the focus of software engineering from manual code production toward higher-level architectural reasoning. Developers may increasingly rely on AI systems to generate implementation details while concentrating on system design, architectural decisions, and validation of generated outputs.

However, the integration of AI technologies into software engineering also introduces significant challenges related to reliability and accountability. One of the most widely discussed concerns is the phenomenon of hallucination in large language models.

Unlike traditional deterministic algorithms, LLMs generate outputs based on probabilistic predictions derived from training data. This process occasionally produces statements that appear plausible but are incorrect.

In scientific writing and technical documentation, hallucinated information can undermine the credibility of generated content. Within software engineering contexts, hallucinated code segments could introduce functional errors, security vulnerabilities, or architectural inconsistencies. Researchers have emphasized that developers must maintain critical oversight when using AI-generated programming suggestions (Alkaissi & Samy, 2023).

Another limitation involves the inherent non-determinism of language model outputs. Studies investigating code generation behavior have shown that identical prompts can yield different outputs depending on sampling parameters and internal model states. While this variability can sometimes produce creative or innovative solutions, it also complicates reproducibility in automated development pipelines (Ouyang, Zhang, Harman, & Wang, 2023).

Security considerations represent an additional dimension of concern. Prompt engineering research has demonstrated that carefully crafted inputs can manipulate model behavior in unintended ways. These vulnerabilities raise important questions about how AI systems should be integrated into secure development environments, particularly when they interact with proprietary codebases (Liu et al., 2023).

Despite these limitations, the potential benefits of AI-assisted software modernization remain substantial. When used responsibly within structured engineering processes, language models can accelerate tasks that traditionally require extensive manual analysis. For example, automated code summarization can dramatically reduce the time required for developers to understand unfamiliar modules.

Similarly, conversational programming interfaces enable developers to explore alternative implementation strategies more efficiently. Rather than manually searching documentation or experimenting with multiple code revisions, engineers can interact directly with AI systems to generate candidate solutions and refine them iteratively.

The integration of AI agents into software development workflows may also facilitate collaborative problem solving across distributed development teams. Because language models can operate through natural language interfaces, they provide a common communication medium that bridges differences in programming expertise and domain knowledge.

However, it is essential to recognize that AI-assisted programming does not eliminate the need for rigorous software engineering practices. Code generated by language models must still undergo thorough testing, security auditing, and architectural validation. Human developers remain responsible for ensuring that generated solutions satisfy functional requirements and comply with system constraints.

Future research directions in this field are likely to focus on improving the reliability, transparency, and interpretability of language models in programming contexts. Advances in model training methodologies may reduce hallucination rates and improve the accuracy of generated outputs. Additionally, hybrid approaches that combine language models with traditional static analysis tools may provide more robust support for architectural refactoring tasks.

Another promising research direction involves the development of specialized AI agents designed specifically for software modernization. These agents could integrate multiple analytical techniques, including dependency analysis, code summarization, and service decomposition algorithms, to provide comprehensive assistance during microservices migration projects.

Ultimately, the transformation of monolithic systems into microservices architectures represents not only a technical challenge but also an organizational and cultural shift within software engineering teams. AI-assisted tools have the potential to support this transformation by reducing technical barriers and enabling developers to navigate complex legacy systems more effectively.

CONCLUSION

The modernization of legacy software systems represents one of the most critical challenges facing contemporary software engineering. As organizations increasingly adopt microservices architectures to support scalable and flexible application development, the need to transform existing monolithic systems has become a central focus of both academic research and industrial practice. However, the complexity of monolithic codebases, combined with limited documentation and tightly coupled dependencies, makes architectural migration a difficult and resource-intensive process.

This research article has explored the emerging role of large language models in supporting software architectural transformation. By synthesizing insights from studies on AI-assisted programming, microservices extraction methodologies, and language model capabilities, the article has provided a comprehensive

theoretical framework for understanding how AI technologies can assist in the modernization of legacy systems.

The analysis demonstrates that large language models possess several capabilities that are highly relevant to software engineering tasks. These include the ability to generate natural language explanations of programming code, produce automated documentation, assist with code generation, and support interactive problem-solving through conversational interfaces. Such capabilities can significantly enhance developer productivity, particularly during tasks that involve analyzing complex codebases and identifying architectural patterns.

At the same time, the study has emphasized that AI-assisted development tools are not a complete replacement for human expertise. Challenges such as hallucination, non-deterministic behavior, and security vulnerabilities highlight the importance of maintaining rigorous validation processes and human oversight when integrating AI technologies into software engineering workflows.

The findings suggest that the most effective approach to AI-assisted software modernization involves a collaborative model in which human developers and intelligent systems work together. In this model, language models function as supportive analytical tools that augment human reasoning rather than autonomous agents that replace traditional engineering practices.

Future research will likely continue to explore new ways of integrating artificial intelligence into the software development lifecycle. As language models become more sophisticated and specialized for programming tasks, they may play an increasingly important role in automating aspects of architectural design, code analysis, and system refactoring.

Ultimately, the convergence of artificial intelligence and software engineering represents a transformative development that has the potential to reshape how complex software systems are designed, maintained, and evolved. By leveraging the strengths of both human expertise and machine intelligence, organizations may achieve more efficient and effective pathways for modernizing legacy software infrastructures.

REFERENCES

1. Ahmadvand, M., & Ibrahim, A. (2016). Requirements reconciliation for scalable and secure microservice (de)composition. Proceedings of the IEEE International Requirements Engineering Conference Workshops.
2. Alkaissi, H., & Samy, M. F. (2023). Artificial hallucinations in ChatGPT: Implications in scientific writing. Cureus, 15(2), e35179.
3. Denny, P., Kumar, V., & Giacaman, N. (2022). Conversing with Copilot: Exploring prompt engineering for solving CS1 problems using natural language. arXiv preprint arXiv:2210.15157.
4. Döderlein, J. B., Acher, M., Khelladi, D. E., & Combemale, B. (2022). Piloting Copilot and Codex: Hot temperature, cold prompts, or black magic? arXiv preprint arXiv:2210.14699.
5. Dong, Y., Xue, J., Jin, Z., & Li, G. (2023). Self-collaboration code generation via ChatGPT. arXiv preprint arXiv:2304.07590.
6. Fritzsch, J., Bogner, J., Zimmermann, A., & Wagner, S. (2018). From monolith to microservices: A classification of refactoring approaches. Proceedings of the International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment.
7. Gysel, M., Kolbener, L., Giersche, W., & Zimmermann, O. (2016). Service cutter: A systematic approach to service decomposition. European Conference on Service-Oriented and Cloud Computing.
8. K. S. Hebbar, “An AI-Augmented Framework for Refactoring Enterprise Monolithic Systems,” INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING, vol. 11, no.8s, pp. 593-604, July. 2023 https://www.ijisae.org/index.php/IJISAE/article/view/8046/7054
9. Knoche, H., & Hasselbring, W. (2018). Using microservices for legacy software modernization. IEEE Software, 35(3), 44-49.
10. Li, S., Zhang, H., Jia, Z., Li, Z., Zhang, C., Li, J., Gao, Q., Ge, J., & Shan, Z. (2019). A dataflow-driven approach to identifying microservices from monolithic applications. Journal of Systems and Software, 157, 110380.
11. Liu, Y., Deng, G., Xu, Z., et al. (2023). Jailbreaking ChatGPT via prompt engineering: An empirical study. arXiv preprint arXiv:2305.13860.
12. Mazlami, G., Cito, J., & Leitner, P. (2017). Extraction of microservices from monolithic software architectures. IEEE International Conference on Web Services.
13. Ouyang, S., Zhang, J. M., Harman, M., & Wang, M. (2023). LLM is like a box of chocolates: The non-

determinism of ChatGPT in code generation. arXiv preprint arXiv:2308.02828.

14. Poldrack, R. A., Lu, T., & Beguš, G. (2023). AI-assisted coding: Experiments with GPT-4. arXiv preprint arXiv:2304.13187.

15. Ren, Z., Wang, W., Wu, G., Gao, C., Chen, W., Wei, J., & Huang, T. (2018). Migrating web applications from monolithic structure to microservices architecture. Asia-Pacific Symposium on Internetware.

16. Sun, W., Fang, C., You, Y., et al. (2023). Automatic code summarization via ChatGPT: How far are we? arXiv preprint arXiv:2305.12865.

17. Xi, Z., Chen, W., Guo, X., et al. (2023). The rise and potential of large language model based agents: A survey. arXiv preprint arXiv:2309.07864.

18. Zhao, W. X., Zhou, K., Li, J., et al. (2023). A survey of large language models. arXiv preprint arXiv:2303.18223.