NORTHWESTERN UNIVERSITY

Automating Microservice Development using Large Language Models

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

MASTER OF SCIENCE

Field of Computer Science

By

Conor Kotwasinski

EVANSTON, ILLINOIS

June 2024

ABSTRACT

Automating Microservice Development using Large Language Models

Conor Kotwasinski

This thesis explores the use of large language models (LLMs) to automatically generate code for microservices, enabling the creation of larger scale software projects through high-level orchestration. By leveraging the capabilities of LLMs to understand natural language descriptions and generate appropriate code, this approach has the potential to significantly improve developer productivity and reduce the complexity of building distributed systems. This work aims to shed light on the feasibility and benefits of LLM-driven microservice development, an important yet under-explored area at the intersection of artificial intelligence and software engineering.

Acknowledgements

I would like to extend my deepest gratitude to my advisors, Professor Zach Wood-Doughty and Professor Xinyu Xing, for their unwavering support, invaluable insights, and expert guidance throughout the course of this research. Their dedication and enthusiasm have been instrumental in shaping this thesis and helping me grow as a researcher.

I am incredibly grateful to my girlfriend, Aimee Morrissey, whose unwavering love, encouragement, and support have been a constant source of motivation throughout the process of completing this thesis. Her belief in me has been a driving force in pursuing my academic goals.

I would also like to express my sincere appreciation to Hongyi Zhou for his insightful ideas and thought-provoking discussions, which have contributed significantly to the development of this research.

Lastly, I would like to thank my family and friends for their understanding, patience, and support throughout my academic journey. Their encouragement has been invaluable in helping me overcome challenges and reach this milestone.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1. Problem Statement

Developing large-scale software systems using microservices architecture can be complex and time-consuming, requiring significant effort in designing, implementing, and integrating individual services. This complexity often leads to increased development costs, longer time-to-market, and potential inconsistencies across the system.

1.2. Significance of the Problem

Addressing the challenges in microservice development is crucial for enabling the rapid creation of scalable, maintainable, and evolvable software systems. Improved approaches to microservice development can lead to increased productivity, reduced costs, and faster innovation in various domains, including web applications, enterprise systems, and cloud-native architectures.

1.3. Microservices Architecture

1.3.1. Definition and Principles

Microservices architecture is an approach to designing software systems as a collection of small, independently deployable services that communicate through well-defined APIs. Each microservice focuses on a specific business capability and can be developed, deployed, and scaled independently [59, 19].

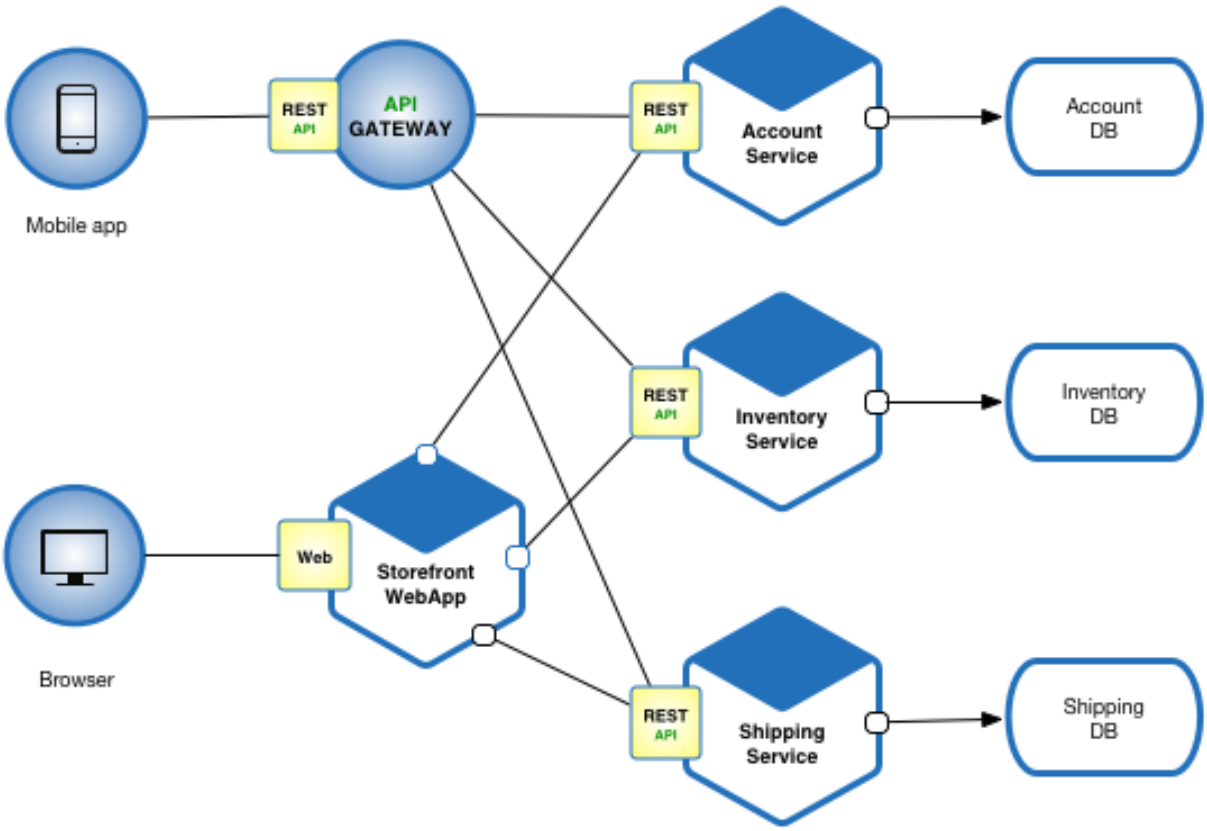

Figure 1.1. Microservice Architecture demonstrating the interaction between services [54].

1.3.2. Benefits of Microservices

Microservices offer several benefits, such as:

- Modularity and flexibility: Services can be developed and updated independently, allowing for faster iteration and adaptation to changing requirements [19].
- Scalability: Individual services can be scaled based on their specific resource needs, enabling efficient resource allocation [19].

- Technology heterogeneity: Microservices can be implemented using different programming languages, frameworks, and technologies, allowing teams to choose the best tools for each service [59].
- Fault isolation: Failures in one service can be isolated, preventing cascading failures across the entire system [59].

1.3.3. Challenges in Microservice Development

Despite the benefits, microservice development poses several challenges:

- Designing appropriate service boundaries and interfaces [59]
- Implementing consistent and interoperable services [71]
- Ensuring service discoverability and scalability [39]
- Managing complex inter-service dependencies and interactions [92]
- Monitoring and debugging distributed systems [92]

1.4. Leveraging Large Language Models for Microservice Development

This thesis proposes the use of large language models (LLMs) to address the challenges in microservice development. LLMs, with their ability to understand and generate human-like text, can be harnessed to automatically generate code for microservices based on high-level descriptions and orchestration.

1.4.1. Potential Benefits

By leveraging LLMs for microservice development, we aim to achieve the following benefits:

- Improved code consistency: LLMs can generate code that adheres to best practices and coding standards, reducing inconsistencies across services [47].

1.4.2. Research Questions

This thesis aims to address the following research questions:

(1) How can LLMs be effectively utilized for generating code for microservices?
(2) What are the key challenges and limitations of LLM-driven microservice development?
(3) How does the performance and quality of LLM-generated microservices compare to manually developed ones?
(4) What are the best practices for integrating LLMs into the microservice development workflow?

1.5. Related Work

1.5.1. Microservices Architecture and Development

Microservices architecture has gained significant attention in recent years due to its benefits in terms of modularity, scalability, and flexibility [59, 19]. Several studies have explored the challenges and best practices in microservice development, including service decomposition [89, 53, 77], and inter-service communication [71].

1.5.2. Large Language Models and Code Generation

Large language models, such as GPT-3 [9], Codex [13], and PaLM [15], have demonstrated impressive capabilities in natural language understanding and generation. These models have been applied to various tasks, including code generation [13, 47], and program synthesis [3].

1.5.3. Tools and Libraries

This thesis leverages several existing tools and libraries to support the proposed methodology. claude-3-opus-20240229 [38], an AI assistant developed by Anthropic, is used as the primary LLM for code generation and analysis. Consul [17], a distributed service discovery and configuration tool, is employed to facilitate service discovery and orchestration among the generated microservices. The methodology also utilizes programming languages such as Python and frameworks like Flask [29] for implementing the microservices and testing scripts.

1.6. Thesis Overview

This thesis proposes a novel approach to microservice development using large language models (LLMs) to automatically generate code based on high-level descriptions and orchestration. By leveraging the powerful language understanding and generation capabilities of LLMs, this approach aims to simplify the development process, improve code consistency, and enable faster iteration. The thesis will explore the feasibility, benefits, and challenges of this approach through a combination of theoretical analysis, experimental evaluation, and case studies. The thesis is organized as follows:

- Chapter 2 provides a comprehensive background on microservices architecture, large language models, code generation techniques, and related work in AI-assisted microservice development.
- Chapter 3 presents the proposed methodology, including the design decisions, experiment setup, and evaluation metrics. It also discusses the tools and libraries used in the implementation.
- Chapter 4 describes the experimental evaluation of the methodology, including the dataset, results, and analysis. It compares the performance and quality of LLM-generated microservices against manually developed ones.
- Chapter 5 showcases case studies that demonstrate the practical application of the methodology in different domains and scenarios.
- Chapter 6 concludes the thesis by summarizing the key findings, discussing the limitations and future work, and highlighting the potential impact of LLM-driven microservice development on software engineering practices.

CHAPTER 2

Background and Related Work

2.1. Microservices Architecture

Microservices architecture (µService) is an approach to designing software systems as a collection of small, independently deployable services that communicate through well-defined APIs [59]. Each microservice focuses on a specific business capability and can be developed, deployed, and scaled independently.

The principles of microservices architecture are based on three Unix ideas [57]:

- A program should fulfill only one task, and do it well.
- Programs should be able to work together.
- Programs should use a universal interface.

These ideas lead to a reusable component design, supporting modularization. The major point is that services are brought to production independently of each other, which is one of the main differences with most Service-Oriented Architecture (SOA) solutions [65]. Microservices architecture offers several benefits [19, 59]:

- Modularity and flexibility: Services can be developed and updated independently, allowing for faster iteration and adaptation to changing requirements.
- Scalability: Individual services can be scaled based on their specific resource needs, enabling efficient resource allocation.

- Technology heterogeneity: Microservices can be implemented using different programming languages, frameworks, and technologies, allowing teams to choose the best tools for each service.
- Fault isolation: Failures in one service can be isolated, preventing cascading failures across the entire system.

However, microservices architecture also presents challenges [39, 89]:

- Designing appropriate service boundaries and interfaces
- Implementing consistent and interoperable services
- Ensuring service discoverability and scalability
- Managing complex inter-service dependencies and interactions
- Monitoring and debugging distributed systems

Microservices architecture emphasizes lightweight virtual machines, implemented as containers (e.g., Docker) or individual processes [65]. This unbinds dependency on a specific technology, enabling usage of a service-specific infrastructure. Each microservice maintains its own context and perspective over particular data, possibly leading to duplications across services [57]. Unlike SOA, microservices do not have an integration component responsible for service orchestration and prefer choreography [65]. Business processes are embedded in services, and there is no logic in the integration. Thus, microservices themselves are responsible for interaction with others. This provides limited flexibility to design or adjust business processes across the company's IT but allows for independent service management and deployment. Microservices architecture fits well in the context of cloud computing and is often referred to as cloud-native [39, 93]. The key features enabling this

are the individual and automated service deployment, supporting system elasticity and scalability [19].

2.2. Large Language Models

Large language models (LLMs) are an emerging class of AI models that contain hundreds of billions or even trillions of parameters and are trained on massive amounts of text data. Prominent examples of LLMs include GPT-3 [9], PaLM [15], Chinchilla [35], and GPT-4 [85]. These models demonstrate remarkable capabilities in natural language understanding, generation, reasoning, and even code synthesis.

A key advancement that has enabled LLMs is the scaling of model size, training data, and compute [9, 41]. Scaling laws have been proposed that show how performance improves predictably as these factors are increased [35, 41]. This has motivated the development of ever larger models, with the latest GPT-4 model from OpenAI and PaLM model from Google containing over 500 billion parameters [15, 85]. In addition to model scale, LLMs are characterized by the emergence of new capabilities that are not present in smaller models, such as in-context learning, instruction following, and step-by-step reasoning [84]. These emergent abilities allow LLMs to perform tasks given only a few demonstrations or examples, without requiring task-specific fine-tuning. LLMs have shown impressive performance across a wide range of benchmarks testing knowledge, reasoning, language understanding and generation [33, 79, 74]. On coding tasks specifically, models like OpenAI Codex [13] and DeepMind AlphaCode [47] have demonstrated the ability to generate code solutions that pass difficult programming challenges with performance competitive with human programmers.

The advent of LLMs is revolutionizing the fields of natural language processing and artificial intelligence more broadly. They provide a foundation for building AI systems that can engage in open-ended dialogue, answer questions, write code, and assist with all sorts of language-related tasks. Techniques for efficiently adapting LLMs to new tasks and ensuring their safety and robustness are active areas of research [63, 4].

2.3. Code Generation with LLMs

There is a rich variety of benchmarks available to evaluate LLMs in code generation tasks. Some commonly used benchmarks include HumanEval [13], DS-1000 [44], and MBPP [3]. Existing work has explored various techniques for using LLMs in code generation. For instance, Codex [13] is a GPT-3 model fine-tuned on publicly available code from GitHub, achieving strong performance on the HumanEval benchmark. Zhang et al. [90] propose a novel Transformer decoding algorithm called Planning-Guided Transformer Decoding (PG-TD) that uses a planning algorithm to guide the Transformer in generating better code. Their empirical evaluation shows that PG-TD generates programs with higher pass rates compared to baseline methods.

Rozière et al. [67] introduced Code Llama, a series of LLMs specifically designed for code generation. Code Llama models are built on the Llama 2 architecture and incorporate code data for pretraining. The authors provide extensive experimental evidence showcasing Code Llama's superior performance on code generation benchmarks like HumanEval, MBPP, and APPS.

Evaluation methods for code generation with LLMs typically involve benchmarks that assess the functional correctness of generated code. However, these benchmarks may not

fully capture the model's ability to generate code for specific domains or application scenarios. For example, HumanEval focuses on synthesizing programs from docstrings, while MBPP evaluates the model's capability to fix bugs and performance issues in code snippets.

When it comes to applying LLMs specifically to microservice development, there is a notable gap in the literature. Existing work primarily focuses on general code generation tasks and benchmarks, without considering the unique characteristics and requirements of microservice architectures. Microservices involve distributed systems, inter-service communication, and specific design patterns, which may require specialized evaluation metrics and benchmarks. The PG-TD algorithm proposed by Zhang et al. [90] could potentially be adapted to address the challenges of microservice code generation, but further research is needed to validate its effectiveness in this domain.

To bridge this gap, future research should explore the development of LLMs tailored for microservice code generation. This may involve fine-tuning LLMs on microservice-specific codebases, incorporating domain knowledge about microservice architectures, and designing evaluation benchmarks that assess the model's ability to generate code adhering to microservice best practices and patterns. Additionally, investigating techniques to handle the context window limitations and enable LLMs to navigate complex microservice codebases effectively would be valuable.

While code generation with LLMs has made significant progress, there are still limitations and challenges to address. Applying LLMs to microservice development remains an under-explored area, requiring further research to develop specialized models, techniques, and evaluation methods that cater to the unique characteristics of microservice

architectures. The work by Zhang et al. [90] on planning-guided code generation provides a promising direction for future research in this domain.

2.4. Specialized Language Models for Code

As pretrained Transformers such as GPT and BERT achieved remarkable success in natural language processing, such model architectures, learning paradigms, and training objectives were soon adopted by the software engineering community to produce specialized models for code understanding and generation [37, 25, 48]. These specialized code language models (Code LMs) can be categorized based on their architecture: encoder-only models [40, 21, 31, 80], encoder-decoder models [82, 61, 11, 10, 83], decoder-only models [76, 49, 60, 16, 91, 30, 18], UniLM [32], and diffusion models [70]. Recent advancements in NLP, such as instruction tuning [50, 55, 83] and reinforcement learning [45, 69], have also been applied to code processing, further enhancing the performance and adaptability of Code LMs. These specialized models have demonstrated impressive results on various code-related tasks, such as code completion, code translation, and code synthesis.

2.5. LLMs in Software Development

The integration of large language models (LLMs) into software development workflows has opened up new possibilities for AI-assisted development. LLMs have been extended with coding tools, such as interpreters [78, 68, 26, 14, 75], execution emulators, and interactive code generation and refinement systems [81, 7, 12, 43, 51, 88]. These extensions allow LLMs to provide more accurate and contextually relevant code suggestions and assist developers in various stages of the development process. Moreover, LLMs have been integrated into popular AI code assistants, such as GitHub Copilot, which offers

features like code generation, vulnerability detection, and license management. IDEs like CodeFuse [18] also leverage LLMs to provide code generation, translation, commenting, and test case generation capabilities. As LLMs continue to advance, building applications on top of them is becoming a crucial task. Open-source frameworks like LangChain, AutoGPT, and WorkGPT provide abstractions over language models for developers, actively revolutionizing the entire software development process.

2.6. Context Windows in LLMs

Context windows play a crucial role in the ability of large language models (LLMs) to generate coherent and contextually relevant code. The context window determines the amount of input text that the model can attend to when generating a response. Larger context windows allow LLMs to consider more information, enabling them to understand and generate code that is consistent with the broader context of the programming task. Recent advancements in LLMs have led to significant increases in context window sizes. For example, GPT-3.5 started with a 4k token context window in November 2022, and within a year, GPT-4 Turbo boasted a 128k token context window in November 2023, representing a 32x increase [85]. This expansion of context windows has important implications for code generation tasks.

However, despite these improvements, the current context window sizes may still be insufficient for many practical coding tasks, especially when dealing with large codebases. This limitation makes it challenging for LLMs to effectively navigate and understand complex codebases without additional techniques like retrieval augmented generation (RAG) [46].

Furthermore, as the input prompt approaches the maximum context window size, the performance of LLMs may degrade. This observation highlights the need for careful management of input prompts to ensure the model can effectively utilize the available context.

Looking forward, significant increases in context window sizes (e.g., 100 million tokens) could potentially enable LLMs to process entire codebases without the need for additional techniques like RAG [66]. However, even with larger context windows, challenges related to reasoning, instruction following, and avoiding hallucinations will likely persist and require further research and development [52].

2.7. Retrieval Augmented Generation for Code LLMs

Retrieval augmented generation (RAG) is a promising approach to address the context window limitations of LLMs in code generation tasks. RAG involves retrieving relevant code snippets from an indexed codebase and injecting them into the input prompt, providing the LLM with additional context and information [46].

Several studies have demonstrated the effectiveness of RAG for code generation with LLMs. Their method retrieves relevant code snippets from a large codebase and progressively refines the generated code through multiple retrieval and generation stages. The authors showed that this approach outperforms baseline methods and achieves state-of-the-art performance on various code generation benchmarks.

RAG has also been applied to specific programming languages and domains. For example, Phan et al. [72] developed JavaBERT, a BERT-based model for Java code generation that incorporates retrieval augmentation. JavaBERT retrieves relevant code

snippets from a large Java codebase and uses them to guide the generation process. The authors showed that JavaBERT outperforms baseline methods and achieves state-of-the-art results on Java code generation benchmarks.

While RAG has shown promising results for code generation with LLMs, there are still challenges and limitations to consider. One challenge is the selection of relevant code snippets from the indexed codebase. The quality and relevance of the retrieved snippets directly impact the generated code's quality and coherence. Various retrieval strategies have been proposed, such as TF-IDF [46], dense vector retrieval [42], and learning-to-rank [62], but finding the optimal retrieval method for a given task remains an open research question.

Another challenge is the integration of retrieved code snippets into the generation process. Simply concatenating the retrieved snippets to the input prompt may not always lead to coherent and consistent generated code.

Moreover, the effectiveness of RAG for code generation may vary depending on the specific programming language, domain, and task complexity. While RAG has shown success in general-purpose programming languages like Python and Java, its applicability to domain-specific languages or highly specialized codebases requires further investigation.

In the context of microservice development, RAG could potentially help LLMs navigate and understand complex microservice codebases by retrieving relevant code snippets related to inter-service communication, service discovery, and other microservice-specific patterns. However, adapting RAG techniques to the unique characteristics and challenges of microservice architectures remains an open research area.

Future research directions in RAG for code generation with LLMs include developing more sophisticated retrieval strategies that consider the semantic and structural similarities between code snippets, exploring techniques for effective integration of retrieved information into the generation process, and investigating the applicability of RAG to various programming languages, domains, and task complexities, including microservice development.

2.8. AI-Assisted Microservice Development

The application of AI techniques, particularly large language models (LLMs), to microservice development is an emerging area of research with significant potential. AI-assisted microservice development aims to leverage the capabilities of LLMs to automate and simplify various aspects of microservice design, implementation, and management.

Another area where AI can assist in microservice development is code generation. As discussed in the previous sections, LLMs have shown impressive capabilities in generating code snippets and even entire programs based on natural language descriptions. Applying these code generation techniques to microservice development could significantly reduce the development effort and improve productivity.

Despite the promising applications of AI in microservice development, there are several challenges and considerations to address. One challenge is ensuring the quality, correctness, and security of the AI-generated code and configurations. Rigorous testing, code review, and validation mechanisms are necessary to mitigate potential risks and ensure the reliability of AI-generated microservices.

Another consideration is the integration of AI-generated microservices with existing systems and infrastructures. Microservices often operate within a larger ecosystem, interacting with other services, databases, and external APIs. Ensuring seamless integration and compatibility of AI-generated microservices with the existing environment is crucial for the overall system's stability and performance [92]. Moreover, the explainability and interpretability of AI-assisted microservice development processes are important factors to consider. Developers and stakeholders should be able to understand the rationale behind the AI-generated designs, code, and decisions. Techniques for explaining and visualizing the AI models' outputs and reasoning processes can enhance trust and facilitate effective collaboration between humans and AI in microservice development [36, 28].

Ethical considerations also come into play when applying AI to microservice development. AI models, including LLMs, can exhibit biases and generate outputs that may have unintended consequences [6]. Ensuring fairness, transparency, and accountability in AI-assisted microservice development is crucial to prevent potential harm and maintain ethical standards [22].

Looking forward, the integration of AI techniques with traditional software engineering practices and tools in microservice development presents exciting opportunities. Hybrid approaches that combine the strengths of AI and human expertise can lead to more efficient, reliable, and innovative microservice development processes [1].

Future research directions in AI-assisted microservice development include exploring techniques for ensuring the quality and security of AI-generated code, developing frameworks for seamless integration of AI-generated microservices with existing systems,

advancing explainable AI methods for microservice development, and addressing ethical considerations in AI-driven software engineering.

As AI technologies continue to evolve and mature, their impact on microservice development is expected to grow. Leveraging the capabilities of AI, particularly LLMs, has the potential to revolutionize the way microservices are designed, implemented, and managed, enabling faster, more efficient, and more intelligent development processes. However, realizing the full potential of AI in microservice development requires ongoing research, collaboration between AI and software engineering communities, and careful consideration of the associated challenges and ethical implications.

2.8.1. Narrative Cohesion in LLMs for Microservice Generation

Narrative cohesion plays a crucial role in ensuring the consistency and coherence of code generated by LLMs for microservices. Future research should explore advanced techniques for maintaining narrative cohesion, such as graph-based representation learning [86].

2.8.2. Consistency in Context for LLM-based Microservice Development

Consistency in context is crucial for the quality and functionality of LLM-generated microservices. Techniques like retrieval augmented generation (RAG) [46] have been proposed to mitigate this limitation by retrieving relevant code snippets and incorporating them into the generation process.

2.8.3. Limitations of Code Llama for Microservice Development

While Code Llama [67] has demonstrated impressive performance in code generation tasks, its applicability to microservice development requires further investigation. The specific limitations of Code Llama in generating code for microservices, such as handling inter-service communication and managing shared resources, need to be explored. Comparative studies between Code Llama and other LLMs in the context of microservice development can provide insights into its strengths and weaknesses. Additionally, research should focus on adapting Code Llama's architecture and training process to better suit the unique requirements of microservice development, such as incorporating microservice-specific patterns and best practices.

2.8.4. Evaluation Metrics for Microservice Code Generation

Evaluating the quality and functionality of LLM-generated microservices requires novel metrics and benchmarks that consider the unique characteristics of microservice architectures [**lu2023unified**]. Existing code generation benchmarks, such as HumanEval [13] and MBPP [3], may not fully capture the complexities of microservice development, such as inter-service communication, fault tolerance, and scalability [73]. Developing a comprehensive evaluation framework that assesses various aspects of microservice code generation, such as functional correctness, performance, maintainability, and security, is crucial for advancing research in this area [47]. Future work should focus on creating standardized benchmarks and datasets specifically designed for evaluating LLM-generated microservices.

2.8.5. Integration of LLMs into Microservice Development Workflows

Integrating LLMs seamlessly into existing microservice development workflows and tools is essential for their practical adoption [1]. Developing frameworks and methodologies that facilitate the effective incorporation of LLM-based code generation into the development process is crucial. Future research should explore the development of integrated development environments (IDEs) and tools that support LLM-based microservice development, enabling developers to leverage the power of LLMs while maintaining control over the development process.

2.8.6. Integration of LLMs with Microservice Orchestration Frameworks

Integrating LLMs with popular microservice orchestration frameworks like Kubernetes , Docker Swarm [58], or Apache Mesos [34] can enable the automated deployment, scaling, and management of LLM-generated microservices.

Research should focus on developing techniques and tools that facilitate the seamless integration of LLMs with microservice orchestration frameworks.

2.8.7. Cross-Domain Capability and Transition from Non-Code to Code Contexts

The ability of LLMs to transition knowledge and skills from non-code domains to code generation opens up new possibilities for microservice development. LLMs have demonstrated impressive performance in various natural language processing tasks [8]. Investigating the cross-domain capabilities of LLMs and their potential to transition from

non-code to code contexts is an important research direction. By leveraging the knowledge and skills acquired from general language processing tasks, LLMs can potentially generate more contextualized and semantically meaningful code for microservices [21].

Future research should explore techniques for effective knowledge transfer and adaptation from non-code to code contexts in LLMs. This may involve developing novel architectures, training strategies, and transfer learning methods that can capture the shared semantics and structures between natural language and programming languages [87]. Additionally, investigating the potential of LLMs to generate code based on high-level requirements, use cases, or user stories expressed in natural language can revolutionize the way microservices are developed, enabling a more intuitive and accessible development process [47].

The integration of large language models (LLMs) into microservice development presents a promising avenue for enhancing productivity, quality, and innovation. However, realizing the full potential of LLMs in this domain requires addressing several challenges and considerations. These include maintaining narrative cohesion across microservices, ensuring consistency in context, adapting LLMs to the unique requirements of microservice architectures, exploring the potential of LLMs beyond code generation, developing appropriate evaluation metrics, integrating LLMs into existing development workflows, ensuring the security and reliability of generated microservices, seamlessly integrating LLMs with orchestration frameworks, and leveraging cross-domain capabilities of LLMs. Future research should focus on developing advanced techniques for narrative cohesion, context management, and knowledge transfer in LLM-based microservice development. Adapting LLMs like Code Llama to the specific needs of microservices, creating standardized

evaluation frameworks, and developing integrated tools that support LLM-based development are also important research directions. Moreover, addressing security and reliability concerns, exploring the integration of LLMs with orchestration frameworks, and investigating the transition from non-code to code contexts are crucial for the practical adoption of LLMs in microservice development.

As AI technologies continue to evolve, the synergy between LLMs and microservice development is expected to grow, leading to more efficient, intelligent, and innovative software engineering practices. However, realizing this potential requires ongoing research, collaboration between the AI and software engineering communities, and careful consideration of the associated challenges and ethical implications. By addressing these aspects, the integration of LLMs into microservice development can pave the way for a new era of AI-driven software engineering, revolutionizing the way complex software systems are built and maintained.

CHAPTER 3

Methodology

3.1. Overview

This thesis proposes a novel approach to microservice development using large language models (LLMs) to automatically generate code based on high-level user stories. The methodology, illustrated in Figure 3.1, involves decomposing the user story into individual microservice descriptions using an LLM (claude-3-opus-20240229 [38]), generating code for each microservice based on its description, creating a monolithic application that fulfills the user story, developing test scripts to evaluate the functionality and correctness of both the microservices and monolithic application, executing them in a controlled environment, and analyzing the test results to assess the quality of the generated code.

The proposed methodology builds upon existing research in automated microservice identification and decomposition [89, 53, 77], code generation with LLMs [13], and automated test generation [23, 2, 5]. It extends these techniques to the domain of microservice development, leveraging the language understanding and generation capabilities of LLMs to automate the entire process from user story to functional code. The methodology aims to address the challenges and limitations identified in previous approaches, such as the lack of comprehensive evaluation frameworks for microservice code generation [92] and the need for specialized LLMs tailored for microservice development [24].

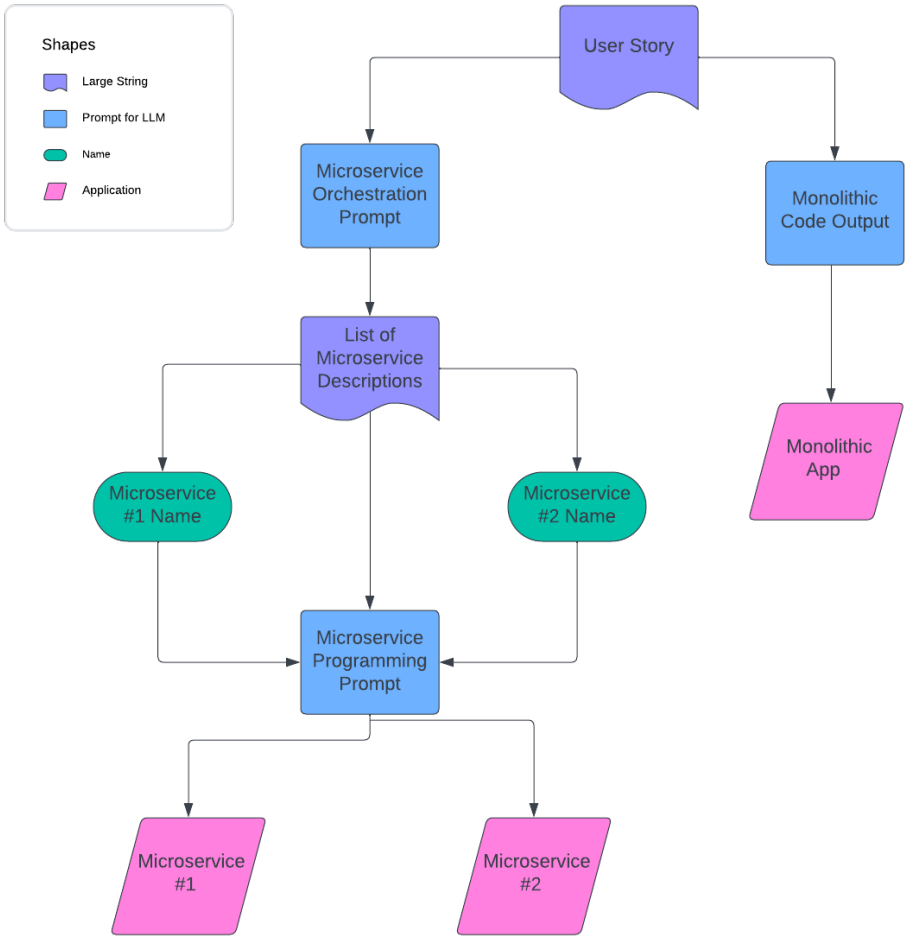

Figure 3.1. Overview of the LLM-based microservice development methodology.

3.2. Distinguishing LLM-Generated and Manual Components

It is important to clearly distinguish between the components generated by the LLMs and those created manually or programmatically in the experimental framework. Figure 3.2 provides an overview of this distinction.

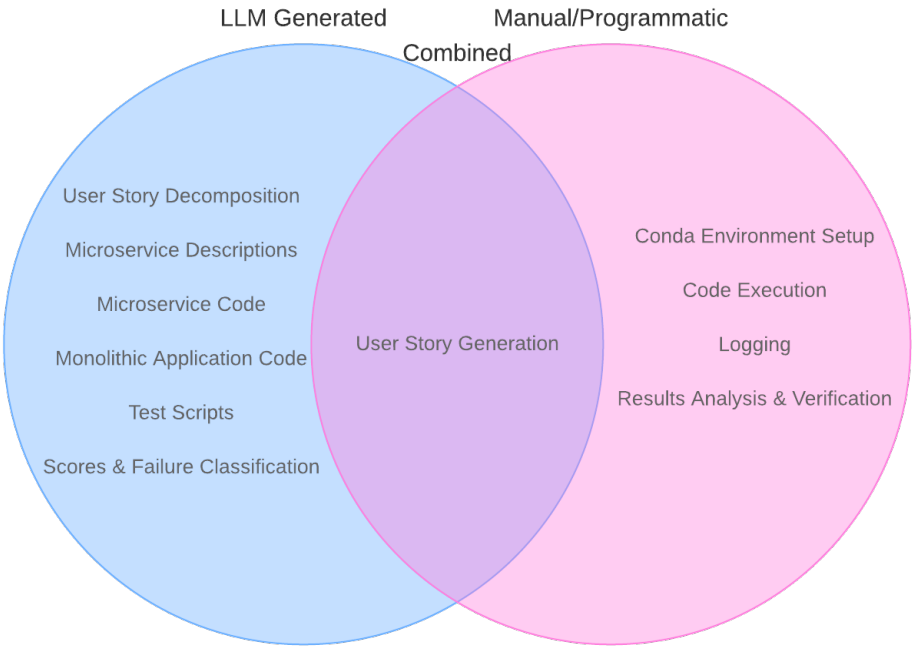

Figure 3.2. This diagram illustrates how tasks are distributed between LLM-generated components (left) and manual or programmatic processes (right), with some overlapping responsibilities in the middle.

The LLMs, specifically claude-3-opus-20240229 [38], is responsible for generating the user story decomposition, microservice descriptions, microservice code, monolithic application code, test scripts, and scores and failure classification based on the test results. These components rely on the language understanding and generation capabilities of the LLMs to automate the development process.

On the other hand, the conda environment setup, code execution, logging, results analysis and verification are handled manually or programmatically. These components ensure a controlled and reproducible experimental setup, facilitate the execution of the generated code, and enable the verified collection and analysis of results. The user stories were generated using a combination of human design and LLM text generation.

3.3. Design Decisions

Several key design decisions were made in the development of this methodology to address common challenges in microservice orchestration and ensure a streamlined experimentation process. Figure 3.3 illustrates the integration of key components such as service discovery and a shared database service within both microservice architecture and a monolithic application.

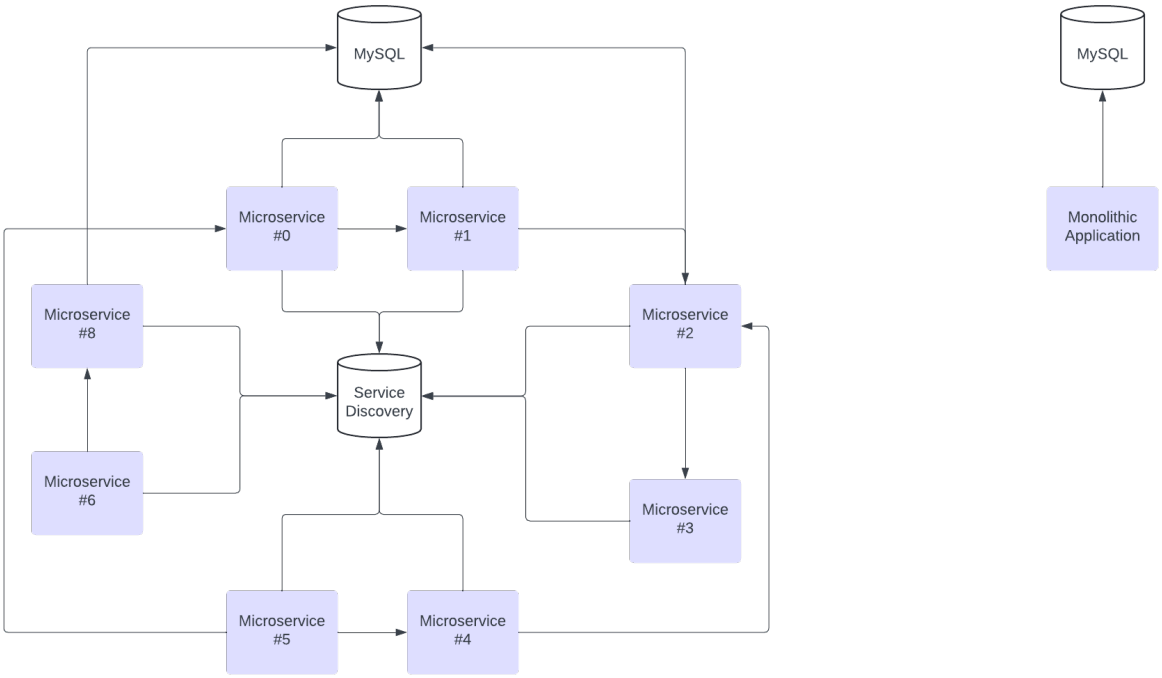

Figure 3.3. Integration of service discovery and shared database service in the microservice architecture and monolithic application

3.3.1. Service Discovery

One of the critical design decisions was to provide a service discovery mechanism (Consul [17]) for the generated microservices. In a microservice architecture, services need to be

able to locate and communicate with each other dynamically [59]. However, the generated microservices had difficulty orchestrating the address and port of each other when running locally. To overcome this challenge, a service discovery mechanism was introduced, allowing microservices to register themselves and discover other services using service names instead of hardcoded addresses and ports. This decision simplifies the orchestration process and enables seamless communication between microservices, as depicted in Figure 3.3.

3.3.2. Shared Database Service

Another important design decision was to offer a shared MySQL database service for both the microservices and the monolithic application. During the initial experiments, it was observed that the generated microservices and monolithic application encountered issues when attempting to set up their own databases within their respective scripts. These issues were primarily related to permissions and other configuration complexities. To mitigate these challenges and ensure a consistent data storage solution, a shared MySQL database service was provided. This decision allows the generated code to focus on the core functionality and business logic, while relying on a preconfigured and accessible database service, as shown in Figure 3.3. Each microserivce cluster or monolithic application is provided an individual database created at runtime to segregate tables from different applications from conflicting with one another. Each database name is supplied with the prompts to develop each application.

3.3.3. Controlled Execution Environment

To ensure the reproducibility and consistency of the experiments, a controlled execution environment was set up using a conda virtual environment. The necessary dependencies, such as Flask [29], requests, and MySQL libraries, were installed within this environment. This decision helps to isolate the runtime dependencies and avoid conflicts with other installed packages on the host system. By executing the microservices, monolithic application, and test scripts within this controlled environment, the methodology ensures that the results are not influenced by external factors and can be easily replicated across different systems.

3.3.4. Automated Test Generation

Evaluating the functionality and correctness of the generated code is a critical aspect of the methodology. To streamline the testing process, the LLM (claude-3-opus-20240229 [38]) was utilized to generate test scripts based on the user story. This decision leverages the language understanding capabilities of the LLM to create comprehensive test cases that align with the desired functionality. The generated test scripts are then adapted to communicate with the microservices and monolithic application endpoints, ensuring thorough testing of the generated code, as detailed in Figure 3.4. Automated test generation reduces manual effort and enables consistent evaluation across different experiments.

3.3.5. Failure Classification

To gain insights into the strengths and weaknesses of the code generation process, the methodology incorporates a failure classification step. The LLM (claude-3-opus-20240229

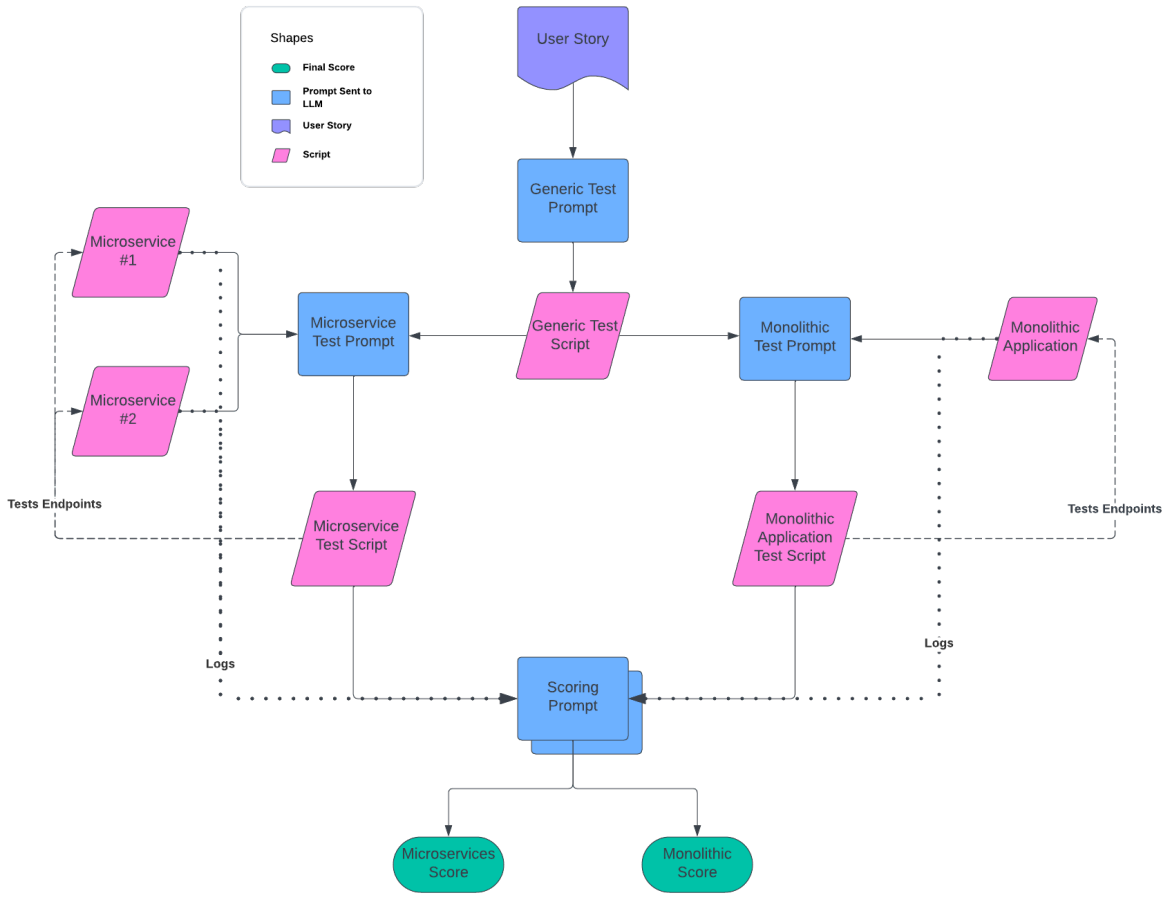

Figure 3.4. Test generation, execution, and scoring process for microservices and monolithic application.

[38]) is used to classify the failed tests into different categories, such as functions not implemented, runtime errors, logic/coding errors, integration failures, configuration failures, and compatibility failures. This decision helps to identify the areas where the LLM struggles to generate correct and complete code. By categorizing the failures, the methodology enables targeted improvements and refinements to the code generation process.

3.4. Experiment Setup

The experiment setup involves several steps to ensure a controlled and reproducible environment for evaluating the LLM-based microservice development approach.

3.4.1. Conda Environment

A conda virtual environment named "thesis_bot" is created with Python 3.9 and the necessary dependencies. The required libraries, such as Flask [29], requests, MySQL [56], and their dependencies, are installed within this environment. The use of a virtual environment ensures a consistent and isolated runtime environment across different experiments and systems.

3.4.2. Microservice and Monolithic Application Execution

The generated microservices and monolithic application are executed simultaneously within the conda environment for a specified duration. This allows for the evaluation of their functionality and performance under realistic conditions. The output logs from the microservices and monolithic application are captured for further analysis.

3.4.3. Test Script Execution

The adapted test scripts for the microservices and monolithic application are executed within the same conda environment. The test results, including the number of passed and failed tests, are captured for each experiment. The test script execution helps to assess the correctness and completeness of the generated code.

3.4.4. Results Analysis

After the execution of the test scripts, the results are analyzed to evaluate the quality of the generated code. The LLM (claude-3-opus-20240229 [38]) is used to classify the failed tests into different categories, providing insights into the common types of failures and areas for improvement. The analysis also includes a summary of the total number of failures in each category and the final test scores for both the microservices and monolithic application.

3.4.5. Execution with Code Llama - Instruct 34B

In addition to claude-3-opus-20240229, the methodology was also attempted with the Code Llama - Instruct 34B model [67]. Code Llama is an open-source foundation model for code, trained on a vast corpus of code and natural language data. It has demonstrated strong performance in various code-related tasks, including code generation, translation, and completion. The execution of the methodology with Code Llama - Instruct 34B was performed on an NVIDIA GeForce RTX 4090 GPU with 24GB of VRAM. To accommodate the model within the available memory, 4-bit quantization was employed. Quantization is a technique that reduces the precision of the model's weights, allowing for more efficient memory usage and faster inference [27].

The 4-bit quantization enables the large Code Llama - Instruct 34B model to be loaded and executed on the RTX 4090 GPU. This quantization approach strikes a balance between model size and performance, allowing for the utilization of powerful language models on consumer-grade hardware. The execution process with Code Llama - Instruct 34B followed the same steps as outlined in the previous subsections, including the conda

environment setup, microservice and monolithic application execution, test script execution, and results analysis. The generated code and test results were compared against those obtained using claude-3-opus-20240229 to assess the performance and quality of the Code Llama model in the context of microservice development.

By incorporating the Code Llama - Instruct 34B model into the methodology, this subsection demonstrates the flexibility and adaptability of the proposed approach to different LLMs. It also highlights the potential for leveraging open-source foundation models and quantization techniques to enable the execution of large language models on consumer-grade hardware, making the methodology more accessible to a wider range of researchers and practitioners.

CHAPTER 4

Experimental Evaluation

Table 4.1. Error Counts for Microservices and Monolithic Architectures (Trial 1)

Error Type	**Microservices**	**Monolithic**
Logic/Coding	19	6
Integration	30	9
Runtime	31	64
Not Implemented	15	30
Configuration	0	5

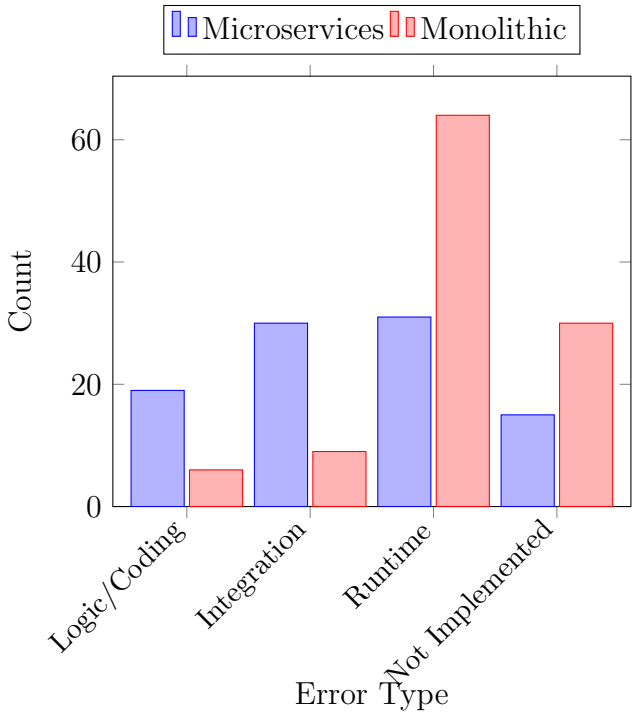

Figure 4.1. Trial 1 Error Comparison

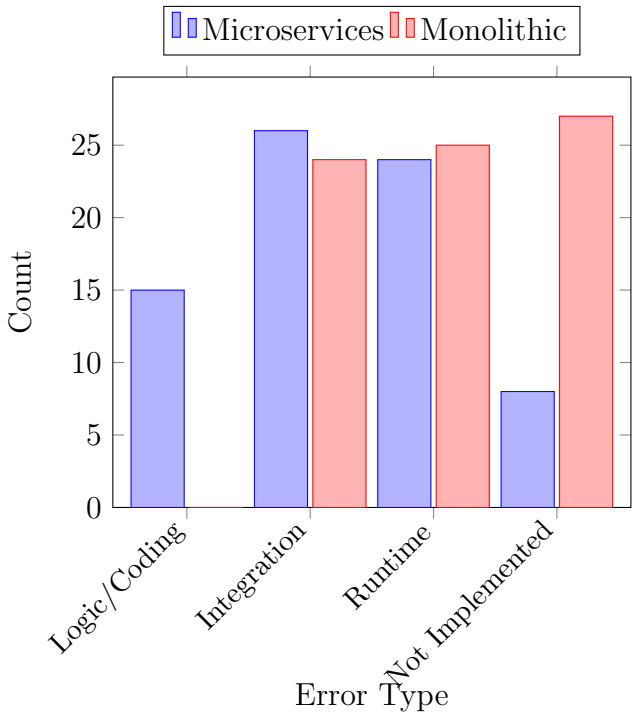

Figure 4.2. Trial 2 Error Comparison

Table 4.2. Error Counts for Microservices and Monolithic Architectures (Trial 2)

Error Type	**Microservices**	**Monolithic**
Logic/Coding	15	0
Integration	26	24
Runtime	24	25
Not Implemented	8	27
Configuration	0	13

The experimental evaluation of the proposed LLM-based microservice development methodology involved testing the generated code for both microservices and monolithic applications across 24 diverse user stories for two trials. The user stories spanned various domains, including customer support ticketing systems, e-commerce platforms, weather forecasting systems, online gaming platforms, and more. For each user story, the LLM

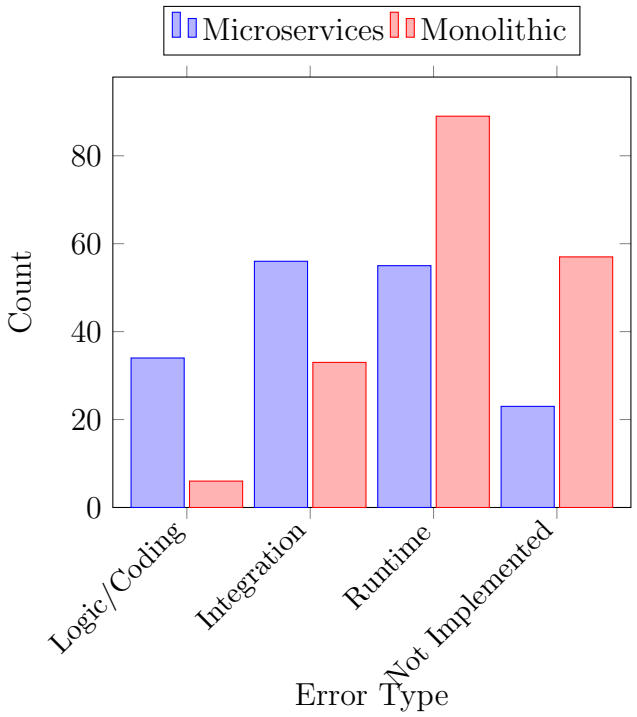

Figure 4.3. Combined Trials Error Comparison

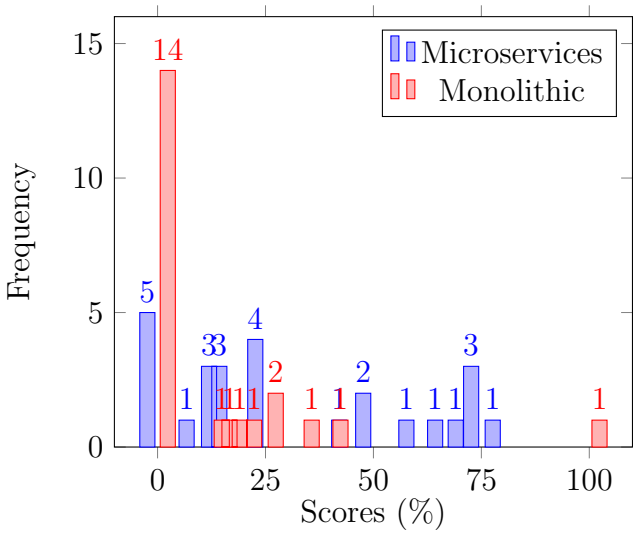

Figure 4.4. Trial 1 scores

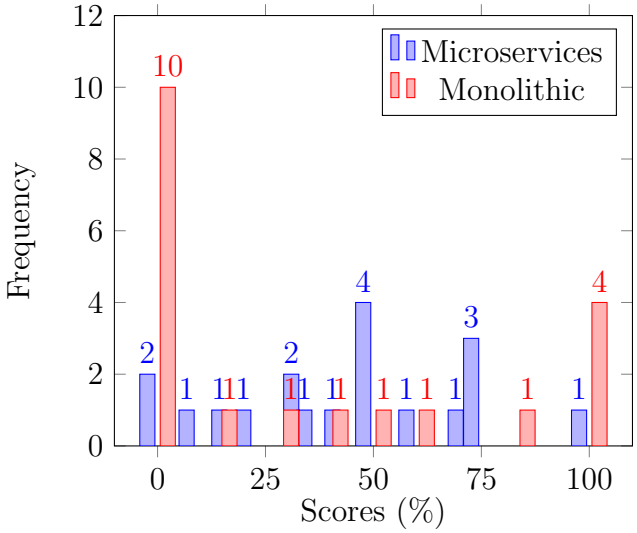

Figure 4.5. Trial 2 scores

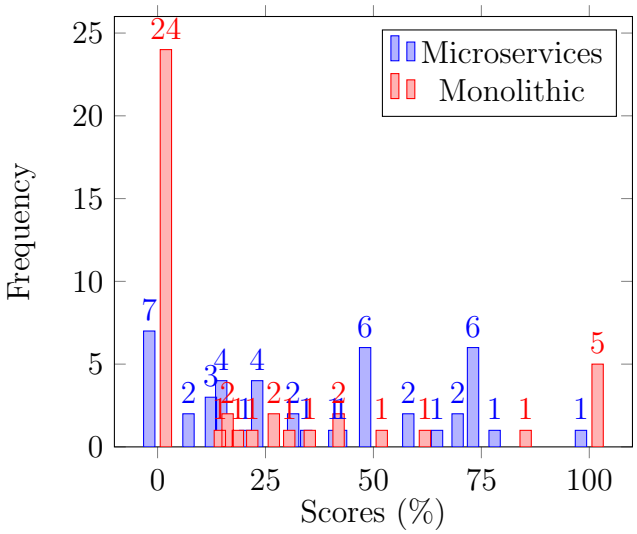

Figure 4.6. Combined trials scores

(claude-3-opus-20240229) was utilized to generate code for the microservices and a corresponding monolithic application. The generated code was then subjected to a series of tests to assess its functionality, correctness, and overall quality.

Table 4.3. Error Counts for Microservices and Monolithic Architectures (Combined)

Error Type	**Microservices**	**Monolithic**
Logic/Coding	34	6
Integration	56	33
Runtime	55	89
Not Implemented	23	57
Configuration	0	18

The evaluation process involved classifying the test failures into six categories: functions not implemented, runtime errors, logic/coding errors, integration failures, configuration failures, and compatibility failures. This categorization provided insights into the strengths and weaknesses of the code generation process and helped identify areas for improvement.

4.1. Results Analysis

The experimental evaluation conducted in this study aimed to compare the performance of microservices and monolithic architectures in the context of LLM-generated code. The results from the two trials provide valuable insights into the strengths and weaknesses of each architectural approach. In Trial 1, the microservices architecture achieved a higher mean score of 30.74% compared to the monolithic architecture's mean score of 12.47%. The median scores further highlight this difference, with microservices having a median of 25.00% and monolithic architecture having a median of 0.00%. These results suggest that the microservices architecture generally performed better in terms of functionality and correctness. However, it is important to note that the standard deviation of scores for microservices (27.98%) was higher than that of the monolithic architecture (21.67%), indicating a greater variation in the performance of individual microservices.

Trial 2 exhibited a similar trend, with microservices achieving a higher mean score (43.33%) and median score (42.90%) compared to the monolithic architecture (32.66% and 14.29%, respectively). The standard deviation of scores for microservices (27.84%) was lower than that of the monolithic architecture (38.59%), suggesting a more consistent performance across microservices in this trial.

The total number of errors and failures encountered during the experiments provides further insights into the robustness of each architectural approach. In Trial 1, microservices encountered a total of 95 errors/fails, while the monolithic architecture encountered 114 errors/fails. Trial 2 showed a similar pattern, with microservices encountering 73 errors/fails and the monolithic architecture encountering 89 errors/fails. These results indicate that microservices architecture may be more resilient to failures compared to the monolithic architecture.

The two-proportion z-test conducted for 'not implemented' errors reveals interesting findings. In Trial 1, the z-test resulted in a z-value of -1.84 and a p-value of 0.0653, indicating a marginally significant difference between the proportions of 'not implemented' errors in microservices and monolithic architectures. However, in Trial 2, the z-test yielded a z-value of -2.98 and a p-value of 0.0029, suggesting a statistically significant difference in the proportions of 'not implemented' errors between the two architectures. This finding supports the initial hypothesis that the monolithic architecture would have a significantly higher number of 'not implemented' errors due to the limitations of the context window.

The dispersion measures, including standard deviation and interquartile range (IQR), provide insights into the variability of scores within each architectural approach. In both trials, microservices exhibited higher IQRs (50.00% in Trial 1 and 44.61% in Trial 2)

compared to the monolithic architecture (20.00% in Trial 1 and 55.00% in Trial 2). This suggests that the performance of individual microservices was more varied, while the monolithic architecture had a more concentrated distribution of scores.

The Mann-Whitney U test was employed to compare the overall scores between microservices and monolithic architectures. In Trial 1, the test resulted in a U-value of 437.0 and a p-value of 0.0118, indicating a statistically significant difference in scores between the two architectures. However, in Trial 2, the U-value of 328.5 and p-value of 0.1562 suggest that the difference in scores was not statistically significant. These results highlight the variability in the performance of the architectures across different trials and emphasize the need for further investigation.

4.2. Interpretation and Discussion

The experimental results provide valuable insights into the performance of microservices and monolithic architectures in the context of LLM-generated code. The higher mean and median scores achieved by microservices in both trials suggest that this architectural approach may be more suitable for generating functional and correct code using LLMs. The fault isolation characteristic of microservices, as evidenced by the distribution of score percentages, could be a contributing factor to their better performance.

The initial hypothesis that the monolithic architecture would have a significantly higher number of 'not implemented' errors due to the limitations of the context window was supported by the results of Trial 2. This finding highlights the potential challenges associated with generating code for monolithic architectures using LLMs, particularly when the code complexity exceeds the available context window. Microservices, on the

other hand, may be more amenable to code generation using LLMs due to their modular and focused nature.

The dispersion measures, particularly the higher IQRs for microservices, indicate that the performance of individual microservices was more varied compared to the monolithic architecture. This variability could be attributed to the differences in the complexity and requirements of each microservice. Further investigation into the factors influencing the performance of individual microservices could provide valuable insights for improving the code generation process.

The statistically significant difference in overall scores between microservices and monolithic architectures in Trial 1, as revealed by the Mann-Whitney U test, supports the notion that microservices may be more suitable for LLM-generated code. However, the lack of a statistically significant difference in Trial 2 underscores the need for additional trials and a larger sample size to draw more definitive conclusions. It is important to acknowledge the limitations of this study, such as the use of a controlled execution environment and a shared database service, which may not fully represent real-world deployment scenarios. Future research could explore the performance of LLM-generated code in more diverse and complex environments to assess its generalizability.

Table 4.4. Statistical Measures for Microservices and Monolithic Architectures

Measure	**Trial 1**		**Trial 2**	
	Microservices	**Monolithic**	**Microservices**	**Monolithic**
Mean Score	30.74%	12.47%	43.33%	32.66%
Median Score	25.00%	0.00%	42.90%	14.29%
Standard Deviation of Scores	27.98%	21.67%	27.84%	38.59%
Total Errors/Fails	95	114	73	89

Table 4.4 summarizes the key statistical measures for microservices and monolithic architectures in both trials, providing a concise overview of the performance differences between the two architectural approaches.

The experimental evaluation conducted in this study provides valuable insights into the performance of microservices and monolithic architectures in the context of LLM-generated code. The results suggest that microservices may be more suitable for code generation using LLMs due to their modular nature and fault isolation characteristics. However, further research is needed to validate these findings and explore the factors influencing the performance of LLM-generated code in various deployment scenarios.

4.3. Methodology Impact on Results

It is essential to consider how the chosen methodology may have influenced the experimental results. The use of a controlled execution environment, with a pre-configured conda virtual environment and necessary dependencies, aimed to minimize external factors and ensure a consistent runtime across experiments. However, this setup may not fully represent the diversity and complexity of real-world deployment environments, potentially limiting the generalizability of the findings [20].

The decision to provide a shared MySQL database service for both microservices and monolithic applications simplified the data storage aspect and reduced configuration complexities. While this approach facilitated the experimentation process, it may have masked potential challenges related to database management and scalability that could arise in production scenarios [64].

The automated test generation process, leveraging the LLM to create test scripts based on user stories, introduced an additional layer of abstraction. While this approach streamlined the testing process, it relied on the LLM's ability to generate comprehensive and accurate test cases. Any limitations or biases in the generated tests could have influenced the evaluation results [71].

The failure classification step, where the LLM categorized failed tests into different error types, provided valuable insights into the nature of the failures. However, the accuracy and consistency of this classification process depended on the LLM's understanding of the error messages and its ability to differentiate between various failure modes. Misclassifications or ambiguities in the categorization could have affected the interpretation of the results [19].

4.4. Challenges with Code LLama Instruct 34B

In the pursuit of reproducibility and transparency in the experimental evaluation, an attempt was made to utilize Code LLama Instruct 34B, an open-source language model specifically designed for code generation tasks. The motivation behind using Code LLama Instruct 34B was its open weights, which allow for better reproducibility and scrutiny of the model's performance compared to proprietary models like claude-3-opus-20240229.

However, despite the potential benefits of using an open-source model, several challenges were encountered during the experimentation process. One of the primary issues was the inability to successfully start a service using the code generated by Code LLama Instruct 34B. The model struggled to generate complete and functional code, often resulting in syntax errors, runtime failures, and incomplete implementations. These challenges

underscore the complexities involved in using large language models for code generation tasks, particularly in the context of microservices and monolithic architectures. The performance of Code LLama Instruct 34B may be influenced by various factors, such as the quality and diversity of its training data, the specific prompts and instructions provided, and the intricacies of the target programming languages and frameworks.

While the open nature of Code LLama Instruct 34B is advantageous for reproducibility and transparency, it is important to recognize that proprietary models like claude-3-opus-20240229 may have undergone more extensive fine-tuning and optimization for specific code generation tasks. The closed nature of these models, however, limits the ability to fully understand and reproduce their results.

To address the challenges encountered with Code LLama Instruct 34B, further investigation and experimentation are necessary. This may involve fine-tuning the model, optimizing the prompts and instructions, or adapting the code generation process to better suit the requirements of microservices and monolithic architectures. Collaboration with the open-source community and sharing of findings can contribute to the improvement of open-source language models for code generation tasks.

In the next chapter, we will present a case study that showcases an example of incomplete programming generated by Code LLama Instruct 34B. This case study will provide a detailed analysis of the specific challenges encountered and discuss potential strategies for mitigating these issues in future experiments and real-world applications. By examining the limitations and opportunities associated with open-source language models like Code LLama Instruct 34B, we aim to contribute to the advancement of reproducible and transparent research in the field of code generation using large language models.

CHAPTER 5

Case Studies

I present three case studies that highlight the impact of architectural choices and the limitations of code generation models on the performance and functionality of the resulting applications. These case studies provide insights into the advantages of microservices architecture, particularly in terms of fault isolation and modularity, and the challenges associated with generating complete and functional code for monolithic applications using language models with limited context windows.

5.1. Travel Booking Platform

The first case study focuses on a travel booking platform, where the microservices architecture demonstrated better performance compared to the monolithic application. The test results reveal the benefits of fault isolation in microservices architecture.

In the microservices implementation, two failed tests were categorized as integration failures. Despite these failures, the overall system remained partially functional, allowing some tests to pass. The final test script score for the microservices architecture was 66.67%, with two out of three tests passing.

In contrast, the monolithic application suffered from multiple failures due to missing functionality and logic errors. The test results showed three failed tests categorized as "functions not implemented" and one failed test categorized as a logic/coding error. These

failures in one part of the monolithic application affected the entire system, resulting in a final test script score of 0%.

This case study exemplifies how the fault isolation provided by microservices architecture can lead to a better overall score. Even when failures occur in one service, they do not necessarily bring down the entire system. The modular nature of microservices allows for the isolation of faults and the continued operation of other services.

5.2. Event Ticketing System

The second case study examines an event ticketing system, where the monolithic application failed to implement certain functionality, likely due to the limited context window of the code generation model. The test results highlight the challenges associated with generating complete and functional code for monolithic applications.

In the monolithic implementation, all four tests failed with a 404 status code, indicating that the endpoints for creating events, defining ticket types, purchasing tickets, and user registration were not implemented. The final test script score for the monolithic application was 0%, with none of the tests passing.

The failure analysis suggests that the necessary endpoints and functionality were missing in the monolithic application. This can be attributed to the code generation model running out of context window while generating the code for the monolithic application. In a monolithic architecture, all the functionality is tightly coupled within a single codebase, making it challenging for the model to generate a complete implementation when constrained by a limited context window.

On the other hand, the microservices architecture for the event ticketing system did not have any failures and achieved a final test script score of 100%. The modular nature of microservices allows for better separation of concerns and independent development of individual services. Even if the code generation model runs out of context window for one microservice, it can still generate the other microservices successfully.

This case study emphasizes the advantages of microservices architecture in terms of modularity and the ability to overcome the limitations of code generation models with limited context windows.

5.3. Code Llama Generated Code

The third case study focuses on the code generated by the Code Llama Instruct 34B model, which registered itself in Consul under port 8081 but hosted itself on port 5000. This discrepancy could be attributed to either the model falling out of the context window or its limitations in writing Flask applications. Figure 5.1 presents a snippet of the problematic code generated by Code Llama Instruct 34B.

The code generated by Code Llama Instruct 34B demonstrates the challenges associated with using open-source language models for code generation tasks. Despite the potential benefits of reproducibility and transparency offered by open-source models, the generated code may suffer from syntax errors, runtime failures, and incomplete implementations, as evidenced by the inconsistency in the port numbers shown in Figure 5.1. This case study highlights the complexities involved in using large language models for code generation tasks, particularly in the context of microservices and monolithic architectures. The performance of code generation models may be influenced by factors such

as the quality and diversity of their training data, the specific prompts and instructions provided, and the intricacies of the target programming languages and frameworks.

While the open nature of Code Llama Instruct 34B is advantageous for reproducibility and transparency, proprietary models like claude-3-opus-20240229 may have undergone more extensive fine-tuning and optimization for specific code generation tasks. However, the closed nature of proprietary models limits the ability to fully understand and reproduce their results.

To address the challenges encountered with Code Llama Instruct 34B, further investigation and experimentation are necessary. This may involve fine-tuning the model, optimizing the prompts and instructions, or adapting the code generation process to better suit the requirements of microservices and monolithic architectures. Collaboration with the open-source community and sharing of findings can contribute to the improvement of open-source language models for code generation tasks.

```
 1  import os
 2  from flask import Flask, jsonify, request
 3  from flask_sqlalchemy import SQLAlchemy
 4  from consul import Consul
 5
 6  app = Flask(__name__)
 7
 8  # Set up SQLAlchemy
 9  app.config["SQLALCHEMY_DATABASE_URI"] = "mysql+pymysql://" + \
10      os.environ["SQL_USERNAME"] + ":" + os.environ["SQL_PASSWORD"] + \
11      "@" + os.environ["SQL_HOST"] + "/" + os.environ["SQL_DB"]
12  db = SQLAlchemy(app)
13
14  # Set up Consul
15  consul = Consul(host=os.environ["CONSUL_HOST"],
16                  port=os.environ["CONSUL_PORT"])
17
18  # Model for volunteers table
19  class Volunteer(db.Model):
20      ...
25
26  # Register the microservice with Consul
27  def register_microservice():
28      consul.agent.service.register(name="volunteer-service", port=8081)
29
30  # Volunteer and event routes
31  @app.route("/volunteers", methods=["GET"])
32  def get_volunteers():
33      ...
74
75  @app.route("/volunteers/<int:volunteer_id>/events", methods=["GET"])
76  def get_volunteer_events(volunteer_id):
77      ...
88
89  if __name__ == '__main__':
90      ...
91      app.run()
```

Figure 5.1. Code snippet generated by Code Llama Instruct 34B. Line 28 shows the registration of the microservice with Consul on port 8081, and line 91 shows the application running on the default port 5000.

CHAPTER 6

Conclusion and Future Work

In this thesis, we have explored the use of large language models (LLMs) for automating microservice development. Our experimental evaluation compared the performance of LLM-generated code for microservices and monolithic architectures across diverse user stories, primarily using the claude-3-opus-20240229 model [38]. The results suggest that microservices may be more suitable for code generation using LLMs due to their modular nature and fault isolation characteristics. However, the study also revealed challenges associated with using open-source language models like Code Llama [67] for code generation tasks. The case studies presented in this thesis highlight the advantages of microservices architecture, particularly in terms of fault isolation and modularity. They demonstrate how the modular nature of microservices allows for the isolation of faults and the continued operation of other services, even when failures occur in one service. In contrast, the monolithic architecture suffered from multiple failures due to missing functionality and logic errors, affecting the entire system.

While the experimental evaluation provides valuable insights, it is important to acknowledge the limitations of the current study. The use of simple prompts and a limited set of tasks and test cases may not fully capture the complexity and diversity of real-world microservice development scenarios. Future research should aim to expand the scope of the study by incorporating more sophisticated prompts, a wider range of tasks, and more

comprehensive test cases. This will help to further validate the findings and assess the generalizability of the LLM-based approach to microservice development.

One of the key hypotheses that emerged from this study is the potential impact of context window size on the ability of LLMs to generate functional code. To directly test this hypothesis, future research should conduct controlled experiments with varying context window sizes. By systematically manipulating the context size and evaluating the quality and completeness of the generated code, we can gain a deeper understanding of the relationship between context size and the performance of LLMs in code generation tasks. This investigation will provide valuable insights into the limitations and requirements of LLMs for effective microservice development.

Although the majority of our experiments were conducted using claude-3-opus-20240229, the challenges encountered with Code Llama Instruct 34B in generating complete and functional Flask applications highlight the need for further investigation into the impact of context window size. As discussed in [67], Code Llama models support input contexts of up to 100,000 tokens, which is significantly larger than the context window sizes of many other LLMs. However, the inconsistency in port numbers and the inability to successfully start services suggest that even with this larger context window, Code Llama may still struggle to generate coherent and fully functional code for complex microservice architectures. Future research should explore techniques to mitigate the limitations imposed by context window size, such as incorporating domain-specific knowledge, optimizing prompts, and adapting the code generation process to better suit the unique requirements of microservices. Another important aspect to explore in future work is the fault tolerance of LLM-generated microservices compared to generated monolithic

programs. To test this hypothesis, researchers can design experiments that introduce controlled faults or failures into the generated code and assess the resilience and recovery capabilities of microservices and monolithic architectures. By measuring the impact of failures on the overall system functionality and the ability to isolate and recover from faults, we can gain a better understanding of the fault tolerance benefits of microservices in the context of LLM-generated code.

As an expert in the current state of these tools, it is crucial to discuss the steps needed before this type of code generation can be deployed or commercialized. While LLMs like claude-3-opus-20240229 and Code Llama have shown promising results in code generation tasks, there are still significant limitations and challenges that need to be addressed. Future work should focus on improving the quality and reliability of the generated code by leveraging larger and more diverse training datasets, incorporating domain-specific knowledge, and developing more advanced prompting techniques. The limitations observed in Code Llama's ability to generate Flask applications underscore the importance of ensuring that the training data adequately covers the target programming languages, frameworks, and design patterns. As highlighted in [67], Code Llama models are trained on a diverse dataset including code, natural language related to code, and general natural language. However, the specific coverage of Flask and other web frameworks in the training data may need to be enhanced to improve the model's performance in generating functional web applications. Additionally, rigorous testing and validation processes must be established to ensure the correctness, security, and performance of the generated code in real-world scenarios.

Another important consideration for future work is the failure triaging and classification process. In this study, we counted different program failures and categorized them into various types. However, it is important to acknowledge that some failures may overlap or include others, which can affect the accuracy of the statistics. Future research should explore more sophisticated failure triaging strategies and develop standardized taxonomies for classifying failures in LLM-generated code. This will help to improve the consistency and reliability of failure analysis and provide a more accurate assessment of the strengths and weaknesses of different architectural approaches.

This thesis has demonstrated the potential of LLMs like claude-3-opus-20240229 and Code Llama for automating microservice development and highlighted the advantages of microservices architecture in terms of fault isolation and modularity. However, it is important to recognize the limitations of the current study and the challenges that need to be addressed before this type of code generation can be deployed or commercialized. Future research should focus on expanding the scope of the study, directly testing key hypotheses related to context window size and fault tolerance, improving the quality and reliability of the generated code through enhanced training data and prompting techniques, and establishing rigorous testing and validation processes. By addressing these limitations and building upon the findings of this thesis, we can pave the way for more advanced and practical applications of LLMs in microservice development and software engineering as a whole. The work presented in [67] provides a solid foundation for further research and development in this area, and we anticipate that the Code Llama models, along with other state-of-the-art LLMs like claude-3-opus-20240229, will continue to evolve and improve, enabling more effective and efficient microservice development in the future.

References

[1] Saleema Amershi et al. "Software Engineering for Machine Learning: A Case Study". en. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Montreal, QC, Canada: IEEE, May 2019, pp. 291–300. ISBN: 978-1-72811-760-7. DOI: 10.1109/ICSE-SEIP.2019.00042. URL: https://ieeexplore.ieee.org/document/8804457/ (visited on 05/09/2024).

[2] Saswat Anand et al. "An orchestrated survey of methodologies for automated software test case generation". en. In: *Journal of Systems and Software* 86.8 (Aug. 2013), pp. 1978–2001. ISSN: 01641212. DOI: 10.1016/j.jss.2013.02.061. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121213000563 (visited on 05/09/2024).

[3] Jacob Austin et al. *Program Synthesis with Large Language Models*. en. arXiv:2108.07732 [cs]. Aug. 2021. URL: http://arxiv.org/abs/2108.07732 (visited on 05/09/2024).

[4] Yuntao Bai et al. *Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback*. en. arXiv:2204.05862 [cs]. Apr. 2022. URL: http://arxiv.org/abs/2204.05862 (visited on 05/09/2024).

[5] Earl T. Barr et al. "The Oracle Problem in Software Testing: A Survey". en. In: *IEEE Transactions on Software Engineering* 41.5 (May 2015), pp. 507–525. ISSN: 0098-5589, 1939-3520. DOI: 10.1109/TSE.2014.2372785. URL: http://ieeexplore.ieee.org/document/6963470/ (visited on 05/09/2024).

[6] Emily M. Bender et al. "On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? " en. In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. Virtual Event Canada: ACM, Mar. 2021, pp. 610–623. ISBN: 978-1-4503-8309-7. DOI: 10.1145/3442188.3445922. URL: https://dl.acm.org/doi/10.1145/3442188.3445922 (visited on 05/09/2024).

[7] Zhangqian Bi et al. *Iterative Refinement of Project-Level Code Context for Precise Code Generation with Compiler Feedback*. en. arXiv:2403.16792 [cs]. Apr. 2024. URL: http://arxiv.org/abs/2403.16792 (visited on 05/09/2024).

[8] Rishi Bommasani et al. *On the Opportunities and Risks of Foundation Models*. en. arXiv:2108.07258 [cs]. July 2022. URL: http://arxiv.org/abs/2108.07258 (visited on 05/09/2024).

[9] Tom B. Brown et al. *Language Models are Few-Shot Learners*. en. arXiv:2005.14165 [cs]. July 2020. URL: http://arxiv.org/abs/2005.14165 (visited on 05/09/2024).

[10] Yekun Chai et al. "ERNIE-Code: Beyond English-Centric Cross-lingual Pretraining for Programming Languages". en. In: *Findings of the Association for Computational Linguistics: ACL 2023*. Toronto, Canada: Association for Computational Linguistics, 2023, pp. 10628–10650. DOI: 10.18653/v1/2023.findings-acl.676. URL: https://aclanthology.org/2023.findings-acl.676 (visited on 05/09/2024).

[11] Saikat Chakraborty et al. "NatGen: generative pre-training by "naturalizing" source code". en. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Singapore Singapore: ACM, Nov. 2022, pp. 18–30. ISBN: 978-1-4503-9413-0. DOI: 10.1145/3540250.3549162. URL: https://dl.acm.org/doi/10.1145/3540250.3549162 (visited on 05/09/2024).

[12] Bei Chen et al. "CODET: CODE GENERATION WITH GENERATED TESTS". en. In: (2023).

[13] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. en. arXiv:2107.03374 [cs]. July 2021. URL: http://arxiv.org/abs/2107.03374 (visited on 05/09/2024).

[14] Wenhu Chen et al. *Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks*. en. arXiv:2211.12588 [cs]. Oct. 2023. URL: http://arxiv.org/abs/2211.12588 (visited on 05/09/2024).

[15] Aakanksha Chowdhery et al. *PaLM: Scaling Language Modeling with Pathways*. en. arXiv:2204.02311 [cs]. Oct. 2022. URL: http://arxiv.org/abs/2204.02311 (visited on 05/09/2024).

[16] Fenia Christopoulou et al. *PanGu-Coder: Program Synthesis with Function-Level Language Modeling*. en. arXiv:2207.11280 [cs]. July 2022. URL: http://arxiv.org/abs/2207.11280 (visited on 05/09/2024).

[17] *Consul by HashiCorp*. en. URL: https://www.consul.io/ (visited on 05/09/2024).

[18] Peng Di et al. *CodeFuse-13B: A Pretrained Multi-lingual Code Large Language Model*. en. arXiv:2310.06266 [cs]. Jan. 2024. DOI: 10.1145/3639477.3639719. URL: http://arxiv.org/abs/2310.06266 (visited on 05/09/2024).

[19] Nicola Dragoni et al. *Microservices: yesterday, today, and tomorrow*. en. arXiv:1606.04036 [cs]. Apr. 2017. URL: http://arxiv.org/abs/1606.04036 (visited on 05/09/2024).

[20] Christian Esposito, Aniello Castiglione, and Kim-Kwang Raymond Choo. "Challenges in Delivering Software in the Cloud as Microservices". In: *IEEE Cloud Computing* 3 (Sept. 2016), pp. 10–14. DOI: 10.1109/MCC.2016.105.

[21] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. en. arXiv:2002.08155 [cs]. Sept. 2020. URL: http://arxiv.org/abs/2002.08155 (visited on 05/09/2024).

[22] Luciano Floridi et al. "AI4People—An Ethical Framework for a Good AI Society: Opportunities, Risks, Principles, and Recommendations". en. In: *Minds and Machines* 28.4 (Dec. 2018), pp. 689–707. ISSN: 0924-6495, 1572-8641. DOI: 10.1007/s11023-018-9482-5. URL: http://link.springer.com/10.1007/s11023-018-9482-5 (visited on 05/09/2024).

[23] Gordon Fraser and Andrea Arcuri. "EvoSuite: automatic test suite generation for object-oriented software". en. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. Szeged Hungary: ACM, Sept. 2011, pp. 416–419. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025179. URL: https://dl.acm.org/doi/10.1145/2025113.2025179 (visited on 05/09/2024).

[24] Yu Gan et al. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems". en. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Providence RI USA: ACM, Apr. 2019, pp. 3–18. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304013. URL: https://dl.acm.org/doi/10.1145/3297858.3304013 (visited on 05/09/2024).

[25] Leo Gao et al. *The Pile: An 800GB Dataset of Diverse Text for Language Modeling*. en. arXiv:2101.00027 [cs]. Dec. 2020. URL: http://arxiv.org/abs/2101.00027 (visited on 05/10/2024).

[26] Luyu Gao et al. *PAL: Program-aided Language Models*. en. arXiv:2211.10435 [cs]. Jan. 2023. URL: http://arxiv.org/abs/2211.10435 (visited on 05/09/2024).

[27] Amir Gholami et al. *A Survey of Quantization Methods for Efficient Neural Network Inference*. en. arXiv:2103.13630 [cs]. June 2021. URL: http://arxiv.org/abs/2103.13630 (visited on 05/09/2024).

[28] Leilani H. Gilpin et al. "Explaining Explanations: An Overview of Interpretability of Machine Learning". en. In: *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)*. Turin, Italy: IEEE, Oct. 2018, pp. 80–89. ISBN: 978-1-5386-5090-5. DOI: 10.1109/DSAA.2018.00018. URL: https://ieeexplore.ieee.org/document/8631448/ (visited on 05/09/2024).

[29] Miguel Grinberg. "Flask Web Development". en. In: ().

[30] Suriya Gunasekar et al. *Textbooks Are All You Need*. en. arXiv:2306.11644 [cs]. Oct. 2023. URL: http://arxiv.org/abs/2306.11644 (visited on 05/09/2024).

[31] Daya Guo et al. *GraphCodeBERT: Pre-training Code Representations with Data Flow*. en. arXiv:2009.08366 [cs]. Sept. 2021. URL: http://arxiv.org/abs/2009.08366 (visited on 05/09/2024).

[32] Daya Guo et al. "UniXcoder: Unified Cross-Modal Pre-training for Code Representation". en. In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Dublin, Ireland: Association for Computational Linguistics, 2022, pp. 7212–7225. DOI: 10.18653/v1/2022.acl-long.499. URL: https://aclanthology.org/2022.acl-long.499 (visited on 05/09/2024).

[33] Dan Hendrycks et al. *Measuring Massive Multitask Language Understanding*. en. arXiv:2009.03300 [cs]. Jan. 2021. URL: http://arxiv.org/abs/2009.03300 (visited on 05/09/2024).

[34] Benjamin Hindman et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center". en. In: ().

[35] Jordan Hoffmann et al. *Training Compute-Optimal Large Language Models*. en. arXiv:2203.15556 [cs]. Mar. 2022. URL: http://arxiv.org/abs/2203.15556 (visited on 05/09/2024).

[36] Fred Hohman et al. "Gamut: A Design Probe to Understand How Data Scientists Understand Machine Learning Models". en. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. Glasgow Scotland Uk: ACM, May 2019, pp. 1–13. ISBN: 978-1-4503-5970-2. DOI: 10.1145/3290605.3300809. URL: https://dl.acm.org/doi/10.1145/3290605.3300809 (visited on 05/09/2024).

[37] Hamel Husain et al. *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search*. en. arXiv:1909.09436 [cs, stat]. June 2020. URL: http://arxiv.org/abs/1909.09436 (visited on 05/10/2024).

[38] *Introducing the next generation of Claude*. en. URL: https://www.anthropic.com/news/claude-3-family (visited on 05/09/2024).

[39] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. "Challenges When Moving from Monolith to Microservice Architecture". In: Feb. 2018, pp. 32–47. ISBN: 978-3-319-74432-2. DOI: 10.1007/978-3-319-74433-9_3.

[40] Aditya Kanade et al. *Learning and Evaluating Contextual Embedding of Source Code*. en. arXiv:2001.00059 [cs]. Aug. 2020. URL: http://arxiv.org/abs/2001.00059 (visited on 05/09/2024).

[41] Jared Kaplan et al. *Scaling Laws for Neural Language Models*. en. arXiv:2001.08361 [cs, stat]. Jan. 2020. URL: http://arxiv.org/abs/2001.08361 (visited on 05/10/2024).

[42] Vladimir Karpukhin et al. *Dense Passage Retrieval for Open-Domain Question Answering*. en. arXiv:2004.04906 [cs]. Sept. 2020. URL: http://arxiv.org/abs/2004.04906 (visited on 05/09/2024).

[43] Shuvendu K. Lahiri et al. *Interactive Code Generation via Test-Driven User-Intent Formalization*. en. arXiv:2208.05950 [cs]. Oct. 2023. URL: http://arxiv.org/abs/2208.05950 (visited on 05/09/2024).

[44] Yuhang Lai et al. *DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation*. en. arXiv:2211.11501 [cs]. Nov. 2022. URL: http://arxiv.org/abs/2211.11501 (visited on 05/09/2024).

[45] Hung Le et al. "CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning". en. In: ().

[46] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. en. arXiv:2005.11401 [cs]. Apr. 2021. URL: http://arxiv.org/abs/2005.11401 (visited on 05/09/2024).

[47] Yujia Li et al. "Competition-Level Code Generation with AlphaCode". en. In: *Science* 378.6624 (Dec. 2022). arXiv:2203.07814 [cs], pp. 1092–1097. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.abq1158. URL: http://arxiv.org/abs/2203.07814 (visited on 05/09/2024).

[48] Anton Lozhkov et al. *StarCoder 2 and The Stack v2: The Next Generation*. en. arXiv:2402.19173 [cs]. Feb. 2024. URL: http://arxiv.org/abs/2402.19173 (visited on 05/10/2024).

[49] Shuai Lu et al. *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*. en. arXiv:2102.04664 [cs]. Mar. 2021. URL: http://arxiv.org/abs/2102.04664 (visited on 05/09/2024).

[50] Ziyang Luo et al. *WizardCoder: Empowering Code Large Language Models with Evol-Instruct*. en. arXiv:2306.08568 [cs]. June 2023. URL: http://arxiv.org/abs/2306.08568 (visited on 05/09/2024).

[51] Aman Madaan et al. *Self-Refine: Iterative Refinement with Self-Feedback*. en. arXiv:2303.17651 [cs]. May 2023. URL: http://arxiv.org/abs/2303.17651 (visited on 05/09/2024).

[52] Joshua Maynez et al. *On Faithfulness and Factuality in Abstractive Summarization*. en. arXiv:2005.00661 [cs]. May 2020. URL: http://arxiv.org/abs/2005.00661 (visited on 05/10/2024).

[53] Genc Mazlami, Jurgen Cito, and Philipp Leitner. "Extraction of Microservices from Monolithic Software Architectures". en. In: *2017 IEEE International Conference on Web Services (ICWS)*. Honolulu, HI, USA: IEEE, June 2017, pp. 524–531. ISBN: 978-1-5386-0752-7. DOI: 10.1109/ICWS.2017.61. URL: http://ieeexplore.ieee.org/document/8029803/ (visited on 05/09/2024).

[54] *Microservices Pattern: Microservice Architecture pattern*. URL: http://microservices.io/patterns/microservices.html (visited on 05/09/2024).

[55] Niklas Muennighoff et al. *OctoPack: Instruction Tuning Code Large Language Models*. en. arXiv:2308.07124 [cs]. Feb. 2024. URL: http://arxiv.org/abs/2308.07124 (visited on 05/09/2024).

[56] *MySQL*. URL: https://www.mysql.com/ (visited on 05/09/2024).

[57] Irakli Nadareishvili et al. "Microservice Architecture: Aligning Principles, Practices, and Culture". en. In: ().

[58] Nitin Naik. "Building a virtual system of systems using docker swarm in multiple clouds". en. In: *2016 IEEE International Symposium on Systems Engineering (ISSE)*. Edinburgh, United Kingdom: IEEE, Oct. 2016, pp. 1–3. ISBN: 978-1-5090-0793-6. DOI: 10.1109/SysEng.2016.7753148. URL: http://ieeexplore.ieee.org/document/7753148/ (visited on 05/09/2024).

[59] Sam Newman. "Building Microservices". en. In: ().

[60] Erik Nijkamp et al. *CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis*. en. arXiv:2203.13474 [cs]. Feb. 2023. URL: http://arxiv.org/abs/2203.13474 (visited on 05/09/2024).

[61] Changan Niu et al. *SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations*. en. arXiv:2201.01549 [cs]. May 2022. URL: http://arxiv.org/abs/2201.01549 (visited on 05/09/2024).

[62] Rodrigo Nogueira and Kyunghyun Cho. *Passage Re-ranking with BERT*. en. arXiv:1901.04085 [cs]. Apr. 2020. URL: http://arxiv.org/abs/1901.04085 (visited on 05/09/2024).

[63] Long Ouyang et al. *Training language models to follow instructions with human feedback*. en. arXiv:2203.02155 [cs]. Mar. 2022. URL: http://arxiv.org/abs/2203.02155 (visited on 05/09/2024).

[64] Claus Pahl and Pooyan Jamshidi. "Microservices: A Systematic Mapping Study:" en. In: *Proceedings of the 6th International Conference on Cloud Computing and Services Science*. Rome, Italy: SCITEPRESS - Science and Technology Publications, 2016, pp. 137–146. ISBN: 978-989-758-182-3. DOI: 10.5220/0005785501370146. URL: https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0005785501370146 (visited on 05/10/2024).

[65] Cesare Pautasso et al. "Microservices in Practice, Part 1: Reality Check and Service Design". en. In: *IEEE Software* 34.1 (Jan. 2017), pp. 91–98. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2017.24. URL: https://ieeexplore.ieee.org/document/7819415/ (visited on 05/09/2024).

[66] Fabio Petroni et al. *Language Models as Knowledge Bases?* en. arXiv:1909.01066 [cs]. Sept. 2019. URL: http://arxiv.org/abs/1909.01066 (visited on 05/10/2024).

[67] Baptiste Rozière et al. "Code Llama: Open Foundation Models for Code". en. In: ().

[68] Timo Schick et al. *Toolformer: Language Models Can Teach Themselves to Use Tools*. en. arXiv:2302.04761 [cs]. Feb. 2023. URL: http://arxiv.org/abs/2302.04761 (visited on 05/09/2024).

[69] Parshin Shojaee et al. *Execution-based Code Generation using Deep Reinforcement Learning*. en. arXiv:2301.13816 [cs]. July 2023. URL: http://arxiv.org/abs/2301.13816 (visited on 05/09/2024).

[70] Mukul Singh et al. "CodeFusion: A Pre-trained Diffusion Model for Code Generation". en. In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Singapore: Association for Computational Linguistics, 2023, pp. 11697–11708. DOI: 10.18653/v1/2023.emnlp-main.716. URL: https://aclanthology.org/2023.emnlp-main.716 (visited on 05/09/2024).

[71] Jacopo Soldani, Damian Tamburri, and Willem-Jan Heuvel. "The Pains and Gains of Microservices: A Systematic Grey Literature Review". In: *Journal of Systems and Software* 146 (Sept. 2018). DOI: 10.1016/j.jss.2018.09.082.

[72] Nelson Tavares de Sousa and Wilhelm Hasselbring. *JavaBERT: Training a transformer-based model for the Java programming language*. en. arXiv:2110.10404 [cs]. Oct. 2021. URL: http://arxiv.org/abs/2110.10404 (visited on 05/09/2024).

[73] Akshitha Sriraman and Thomas F. Wenisch. " Suite: A Benchmark Suite for Microservices". en. In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. Raleigh, NC: IEEE, Sept. 2018, pp. 1–12. ISBN: 978-1-5386-6780-4. DOI: 10.1109/IISWC.2018.8573515. URL: https://ieeexplore.ieee.org/document/8573515/ (visited on 05/09/2024).

[74] Aarohi Srivastava et al. *Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models*. en. arXiv:2206.04615 [cs, stat]. June 2023. URL: http://arxiv.org/abs/2206.04615 (visited on 05/09/2024).

[75] Dídac Surís, Sachit Menon, and Carl Vondrick. *ViperGPT: Visual Inference via Python Execution for Reasoning*. en. arXiv:2303.08128 [cs]. Mar. 2023. URL: http://arxiv.org/abs/2303.08128 (visited on 05/09/2024).

[76] Alexey Svyatkovskiy et al. *IntelliCode Compose: Code Generation Using Transformer*. en. arXiv:2005.08025 [cs]. Oct. 2020. URL: http://arxiv.org/abs/2005.08025 (visited on 05/09/2024).

[77] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation". en. In: *IEEE Cloud Computing* 4.5 (Sept. 2017), pp. 22–32. ISSN: 2325-6095. DOI: 10.1109/MCC.2017.4250931. URL: http://ieeexplore.ieee.org/document/8125558/ (visited on 05/09/2024).

[78] Romal Thoppilan et al. *LaMDA: Language Models for Dialog Applications*. en. arXiv:2201.08239 [cs]. Feb. 2022. URL: http://arxiv.org/abs/2201.08239 (visited on 05/09/2024).

[79] Alex Wang et al. *SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems*. en. arXiv:1905.00537 [cs]. Feb. 2020. URL: http://arxiv.org/abs/1905.00537 (visited on 05/09/2024).

[80] Xin Wang et al. *SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation*. en. arXiv:2108.04556 [cs]. Sept. 2021. URL: http://arxiv.org/abs/2108.04556 (visited on 05/09/2024).

[81] Xingyao Wang et al. *LeTI: Learning to Generate from Textual Interactions*. en. arXiv:2305.10314 [cs]. Mar. 2024. URL: http://arxiv.org/abs/2305.10314 (visited on 05/09/2024).

[82] Yue Wang et al. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. en. arXiv:2109.00859 [cs]. Sept. 2021. URL: http://arxiv.org/abs/2109.00859 (visited on 05/09/2024).

[83] Yue Wang et al. *CodeT5+: Open Code Large Language Models for Code Understanding and Generation*. en. arXiv:2305.07922 [cs]. May 2023. URL: http://arxiv.org/abs/2305.07922 (visited on 05/09/2024).

[84] Jason Wei et al. *Emergent Abilities of Large Language Models*. en. arXiv:2206.07682 [cs]. Oct. 2022. URL: http://arxiv.org/abs/2206.07682 (visited on 05/09/2024).

[85] *What is the difference between the GPT-4 model versions? | OpenAI Help Center*. en. URL: https://help.openai.com/en/articles/7127966-what-is-the-difference-between-the-gpt-4-model-versions (visited on 05/09/2024).

[86] Zonghan Wu et al. “A Comprehensive Survey on Graph Neural Networks”. en. In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (Jan. 2021), pp. 4–24. ISSN: 2162-237X, 2162-2388. DOI: 10.1109/TNNLS.2020.2978386. URL: https://ieeexplore.ieee.org/document/9046288/ (visited on 05/09/2024).

[87] Tianbao Xie et al. *UnifiedSKG: Unifying and Multi-Tasking Structured Knowledge Grounding with Text-to-Text Language Models*. en. arXiv:2201.05966 [cs]. Oct. 2022. URL: http://arxiv.org/abs/2201.05966 (visited on 05/09/2024).

[88] John Yang et al. *InterCode: Standardizing and Benchmarking Interactive Coding with Execution Feedback*. en. arXiv:2306.14898 [cs]. Oct. 2023. URL: http://arxiv.org/abs/2306.14898 (visited on 05/09/2024).

[89] Dongjin Yu et al. “A survey on security issues in services communication of Microservices-enabled fog applications”. en. In: *Concurrency and Computation: Practice and Experience* 31.22 (2019). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4436, e4436. ISSN: 1532-0634. DOI: 10.1002/cpe.4436. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4436 (visited on 05/09/2024).

[90] Shun Zhang et al. *Planning with Large Language Models for Code Generation*. en. arXiv:2303.05510 [cs]. Mar. 2023. URL: http://arxiv.org/abs/2303.05510 (visited on 05/09/2024).

[91] Qinkai Zheng et al. *CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X*. en. arXiv:2303.17568 [cs]. Mar. 2023. URL: http://arxiv.org/abs/2303.17568 (visited on 05/09/2024).

[92] Xiang Zhou et al. “Benchmarking microservice systems for software engineering research”. en. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. Gothenburg Sweden: ACM, May 2018, pp. 323–324. ISBN: 978-1-4503-5663-3. DOI: 10.1145/3183440.3194991. URL: https://dl.acm.org/doi/10.1145/3183440.3194991 (visited on 05/09/2024).

[93] Xiang Zhou et al. “Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study”. en. In: *IEEE Transactions on Software Engineering* 47.2 (Feb. 2021), pp. 243–260. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2018.2887384. URL: https://ieeexplore.ieee.org/document/8580420/ (visited on 05/09/2024).

APPENDIX A

Key Elements of the Experimental Code

A.1. Microservice Description Prompt

The microservice description prompt, shown in Listing A.1, is used to generate detailed descriptions of the required microservices based on the provided user story. The prompt specifies the information to be included for each microservice, such as its name, purpose, endpoints, request/response data structures, and interactions with other microservices.

Listing A.1. Microservice description prompt

```
microservice_descriptions_prompt = f"""
Here is a user story: {user_story}. Based on this user story,
   provide a full description of each required microservice,
   including:

Name (surround the name with [MS_NAME] tags, e.g., [MS_NAME]
   UserService[/MS_NAME])
Service name for service discovery
Purpose
Endpoints (with HTTP methods and paths)
Request/response data structures (include sample data)
Connections to other microservices
```

Specific details needed to interact with other microservices (or tables and data formats used in the database)
Port (to avoid overlapping port numbers)

For each microservice description, make sure to:

Clearly specify the request/response data formats for all endpoints, including sample data.
Explicitly state the service names of any connected microservices when describing their interactions.
Specify which microservice creates each database table and provide the necessary SQL CREATE TABLE statements.
Always use the provided service discovery mechanism (Consul) to register and discover microservices.

The following services are available:

Service Discovery (Consul): {service_discovery_url}
SQL Service: {sql_service_url}:{sql_service_port} (Username: {sql_username}, Password: {sql_password}, Database: {sql_database})

```
Use the provided service discovery and SQL service URLs in your
   microservice descriptions, if necessary. Respond with an array
   of strings using triple quotes, where each string provides a
   thorough description of a single microservice required to
   implement the given user story. Include both the sample data
   and the API details, including specifics on their service names
    and the expected data request/response data structures, for
   the other microservices communicated within each description.
"""
```

A.2. Monolithic Application Prompt

The monolithic application prompt, shown in Listing A.2, is used to generate the complete Python code for a monolithic application based on the provided user story. The prompt specifies the requirements for the generated code, such as using the Flask framework, creating necessary database tables, and including error handling.

Listing A.2. Monolithic application prompt

```
monolithic_application_prompt = f"""
Here is a user story: {user_story}. Based on this user story,
   provide the complete Python code for a monolithic application
   that hosts an API.
Make sure to:
1. Use the Flask framework to create the microservice.
```

2. *Create the necessary database tables using the provided SQL CREATE TABLE statements. Ensure that the tables are created in the correct order, so that tables being referenced by foreign keys already exist.*
3. *Include error handling and appropriate HTTP status codes.*
4. *Use the mysql-connector-python library to interact with the MySQL database.*
5. *Do not use environment variables. Store the necessary information within the script.*

Use the following SQL service URL in your monolithic application code, if necessary:

SQL Service: {sql_service_url}:{sql_service_port} (Username: {sql_username}, Password: {sql_password}, Database: {monolithic_sql_database})

The socket path is '/var/run/mysqld/mysqld.sock'

When using the mysql.connector.connect() function and pass the connection parameters as separate arguments.

Make sure to create any necessary tables.

Disable debug mode in Flask by setting debug=False when calling app.run().

Respond with the sample data included in the complete Python code.

```
"""
```

A.3. Microservice Failure Analysis Prompt

The microservice failure analysis prompt, shown in Listing A.3, is used to classify the failed tests for the microservices based on the test script, microservice code, test output, and microservice logs. The prompt provides guidelines for categorizing failures and requests a detailed explanation for each categorization, as well as suggestions for improving the failure triaging strategy.

Listing A.3. Microservice failure analysis prompt

```
microservice_score_prompt = f"""
Here was my test script:
{microservice_test_code}
Here was the code for my microservices:
{microservice_code}
Here was the output of the test script:
{microservice_test_output}
Here was the logs from the microservices:
{combined_output}
When categorizing the failed tests, please follow these guidelines
   :
```

If a failure is due to a function not being implemented, prioritize categorizing it as "functions not implemented" even if it also results in runtime errors or other issues.

If a failure is caused by a runtime error but not due to a missing function, categorize it as a "runtime error".

If a failure is caused by incorrect logic or coding errors, categorize it as "logic/coding errors".

If a failure is caused by issues with integrating different components or microservices, categorize it as "integration failures".

If a failure is caused by incorrect configuration settings, categorize it as "configuration failures".

If a failure is caused by compatibility issues between different components, libraries, or versions, categorize it as " compatibility failures".

If a failure does not fall under any of the above categories, categorize it as "other" and provide a brief explanation.

For each failed test, provide a clear explanation of why you categorized it into a specific failure type. If a failure seems to fit into multiple categories, explain your reasoning for choosing the most appropriate category based on the root cause of the failure.

```
After categorizing the failures, please provide suggestions on how
    the failure triaging strategy could be improved to ensure more
    accurate statistics. Consider discussing any overlaps or
   dependencies between different failure types and propose ways
   to handle such cases to maintain the integrity of the failure
   counts.
In addition to the total number of each type of failure and the
   final test script score, please provide a summary of the
   failure counts and percentages. Calculate the percentage of
   each failure type relative to the total number of tests.
"""
```

A.4. Monolithic Application Failure Analysis Prompt

The monolithic application failure analysis prompt, shown in Listing A.4, is similar to the microservice failure analysis prompt but focuses on categorizing failed tests for the monolithic application.

Listing A.4. Monolithic application failure analysis prompt

```
monolithic_score_prompt = f"""
Here was my test script:
{monolithic_test_code}
Here was the code for my monolithic application:
{monolithic_code}
Here was the output of the test script:
```

{monolithic_test_output}

Here was the logs from the monolithic application:

{monolithic_output}

When categorizing the failed tests, please follow these guidelines:

If a failure is due to a function not being implemented, prioritize categorizing it as "functions not implemented" even if it also results in runtime errors or other issues.

If a failure is caused by a runtime error but not due to a missing function, categorize it as a "runtime error".

If a failure is caused by incorrect logic or coding errors, categorize it as "logic/coding errors".

If a failure is caused by issues with integrating different components or microservices, categorize it as "integration failures".

If a failure is caused by incorrect configuration settings, categorize it as "configuration failures".

If a failure is caused by compatibility issues between different components, libraries, or versions, categorize it as "compatibility failures".

If a failure does not fall under any of the above categories, categorize it as "other" and provide a brief explanation.

```
For each failed test, provide a clear explanation of why you
    categorized it into a specific failure type. If a failure seems
     to fit into multiple categories, explain your reasoning for
    choosing the most appropriate category based on the root cause
    of the failure.
After categorizing the failures, please provide suggestions on how
     the failure triaging strategy could be improved to ensure more
     accurate statistics. Consider discussing any overlaps or
    dependencies between different failure types and propose ways
    to handle such cases to maintain the integrity of the failure
    counts.
In addition to the total number of each type of failure and the
    final test script score, please provide a summary of the
    failure counts and percentages. Calculate the percentage of
    each failure type relative to the total number of tests.
"""
```

APPENDIX B

Conda Environment Setup

To ensure a consistent execution environment for the experiments, a conda virtual environment named "thesis_bot" is set up with Python 3.9 and the following necessary dependencies:

- flask
- requests
- pyjwt
- psycopg2
- json
- pika
- fastapi
- flask-sqlalchemy
- passlib
- python-jose
- sendgrid
- pydantic¡2
- werkzeug
- mysql
- mysql-connector-python

- mysqlclient
- flask-mail
- pymysql
- flask_mysqldb (installed via pip)
- python-consul (installed via pip)
- Flask-SocketIO (installed via pip)
- redis (installed via pip)

These commands create a new conda environment named "thesis_bot" with Python 3.9 and install the required libraries such as Flask, requests, MySQL, and their dependencies. The environment is used to execute the microservices, monolithic application, and test scripts consistently across experiments.