Tampere University

Saurabh Chauhan

LLM-GENERATED MICROSERVICE IMPLEMENTATIONS FROM RESTFUL API DEFINITIONS

Master's thesis
Faculty of Information Technology and Communication Sciences (ITC)
Supervisors: Prof. Pekka Abrahamsson
Res. Zeeshan Rasheed
December 2024

Abstract

Saurabh Chauhan: LLM-GENERATED MICROSERVICE IMPLEMENTATIONS FROM RESTFUL API DEFINITIONS
Master's thesis
Tampere University
Master of Science, Software, Web, and Cloud
December 2024

Microservice architecture has become a widely adopted architecture pattern in the dynamic field of software development because of the need for scalable, maintainable, and quickly deployable systems. This thesis presents a system employing Large Language Models (LLMs) to automate the API-first RESTful microservice development. Our system takes OpenAPI specifications as input, generate the server-side code, and help to iteratively improves it based on the feedback execution log and error outputs. Following API-first approach, this system guarantees that microservices are designed using clear interfaces, facilitating consistency and reliability throughout the development life-cycle. With log analysis incorporated, the LLM can identify issues and can help to resolve them, thus reducing the amount of iterations. It automates the creation of microservices, and makes the debug and refining process easier, so developers can focus on higher-level design and integration tasks. This system can help and support software developers, architects, and organizations to accelerate software development cycles and minimize manual effort. To understand the capabilities of the system, we conducted an survey with six industry practitioners. The system showed significant benefits when it came to speeding up development, automating repetitive tasks, and simplifying prototyping. The code is publicly available at https://github.com/sirbh/code-gen

Keywords: OpenAPI, Artificial Intelligence, Natural Language Processing, Generative AI, Software Engineering, Large Language Model, Microservices, API-First, Design First, REST, RESTFul API

USE OF AI BASED TOOLS IN THIS THESIS

ChatGPT

How It Was Used: ChatGPT played a key role in organizing and refining the content of this thesis. It reviewed the initial table of contents and provided suggestions to make the structure more logical and cohesive. Additionally, it helped improve the clarity of section titles by suggesting more effective wording. Overall, ChatGPT was instrumental in shaping the flow and readability of the document.

Grammarly

How It Was Used: Grammarly was used throughout the thesis to improve the grammar, spelling, and overall clarity of the writing. It provided suggestions for sentence structure, word choice, and punctuation to ensure the content was clear and free from common language errors.

Responsibility and Ethical Considerations

While AI tools contributed to the technical and editorial processes, the author retains full responsibility for the content of this thesis, including sections influenced or generated with AI assistance. Any potential ethical concerns, inaccuracies, or violations of publication standards remain the responsibility of the author.

PREFACE

I want to sincerely thank Zeeshan Rashid for his constant guidance and feedback during this work. His support has been invaluable in shaping this thesis.
I also want to thank GPT Labs for providing the tools and resources that made this research possible.
Lastly, I am grateful to Pekka Abrahamsson for selecting my thesis topic and serving as my supervisor.

Tampere, December 2024
Saurabh Chauhan

List of Figures

2.1 Microservice Architecture 4
2.2 OpenAPI Specification for the User Service 12

3.1 System Design 14
3.2 JSON to directories 16
3.3 Specification Generation Screen 23
3.4 Server Interaction Screen 24
3.5 Specification Generation 25
3.6 Code Generation 26
3.7 Interaction With Server 27

4.1 Data Analysis 32

List of Tables

3.1 LLM Agent Details 18
3.2 Background Information Questions 28
3.3 Feedback Questions for the System 29
3.4 Participants' Years of Experience in Software Development 29
3.5 Participants' Years of Experience in Microservice Architecture 30

Contents

1 Introduction 1

2 Background 3

2.1 Generative AI 3

2.2 LLMs for Code Generation 3

2.3 Microservices Architecture 4

2.3.1 Key Characteristics of Microservices 4

2.3.2 Advantages of Microservices 5

2.3.3 The Need for Microservices 6

2.3.4 Challenges with Microservices 7

2.4 API-First Approach 9

2.4.1 Overview of API-First Approach 9

2.4.2 Role in Microservices Architecture 10

2.4.3 Standardization with OpenAPI Specification 10

2.4.4 Parallel Development and Testing 11

2.4.5 Benefits of API-First Approach 11

3 Research Methodology 13

3.1 Research Questions (RQs) 13

3.2 System Design 13

3.2.1 OpenAPI Specification Generation 15

3.2.2 Finalization of specification 15

3.2.3 Server Code Generation 16

3.2.4 Automated Validation and Execution 17

3.2.5 Iterative Code Fixing 19

3.2.6 Generated Server Code 20

3.3 Implementation 22

3.3.1 Front-end 22

3.3.2 Back-end 24

3.4 Evaluation Framework 27

3.4.1 Questionnaire Design 27

3.4.2 Practitioners Selections 28

3.4.3 Data Collection 30

3.4.4 Data Analysis 30

4 Preliminary Results 31

4.1 RQ1: User Perceptions of Usability and Effectiveness 31

4.2 RQ2: Reduction in Tool Switching 32

4.3 RQ3: Impact on Manual Coding Efforts and Development Speed . . . 33
5 Discussion . . . 34
6 Future Work . . . 35
6.1 Enhancements to the System . . . 35
6.2 Expansion to Other Use Cases . . . 35
References . . . 40
APPENDIX A. Example OpenAPI Specification Created Using System . . . 41
1 OpenAPI Specification for Product Service API . . . 41
APPENDIX B. Example Microservice Code Created Using System . . . 45
1 Server Code . . . 45
2 Package Management File . . . 45
3 Dockerfile Example . . . 46
4 Docker Compose Configuration . . . 46
5 Environment Variables . . . 47
6 Express.js Routes . . . 47
7 Prisma Schema . . . 48
8 Controller File . . . 48

1 Introduction

The organizations have recently taken a deep interest in the microservice architecture [29]. This architecture has gained so much fame because it segments the application components into several smaller independent services, hence scaling, developing, and deploying becomes quite easier [5]. Each microservice is independently designed to serve a particular function, enabling concentrated development. This modularity allows for independent deployability, faster updates, and the flexibility for teams to choose the best tools and languages for each service, optimizing performance and productivity. Additionally, services can be scaled individually based on demand, promoting efficient resource usage. Lastly, this architecture also promotes fault isolation which means failure to one service does not impact the other making the system resilient[25].

Mircoservice architecture promotes flexibility but it introduces several challenges. For example one of the issue is to effectively communicate the changes done in one service to other stakeholders. Breaking changes done in the API of one service can remain unidentified until runtime. Furthermore, development teams often rely on manual communication to notify other stakeholders about API changes which slow down the development process and increase the risk of human error [15]. In order to address these issues, API-First approach can be utilized which focuses on defining API before implementation allowing development teams to create API contracts that specify how a service will interact [24]. Open API specification (OAS) can be used to define these API contracts making it easier for developers and machine to understand and interact with them [20]. Furthermore, their are tools like SwaggerUI that can be used to interact with these OAS files making users to test and explore the API [30]. Lastly, one of the major benefits of OAS is to allow versioning of API contracts helping development teams to change and communicate the updates effectively.

While the OpenAPI specification can provide significant advantages like versioning and effective communication but its adoption introduces certain challenges. Writing these Open API specs requires deep understanding of the OAS format and tools related to it. Development teams might require training or they need to make themselves familiar with syntax and tools related to it so that the created specification document is both accurate and functional [14].

Furthermore, once the API spec has been generated, development teams need to accurately translate it into working code which again increases the risk of human error. This problem can be resolved by using automatic code generators like Swagger Codegen. But this generated code might not follow coding conventions of the

organization and they need to make the necessary modifications. Furthermore, this approach will primarily generate foundational elements such as request and response models, basic structural code, and documentation comments;the core functionality and business logic must be developed and implemented by the development team.

In order to address above mentioned challenges, introducing generative AI in development process is a promising solution. AI driven tools leveraging LLM can assist development teams in writing Open API specifications. Such tools can reduce the learning curve significantly, since developer can generate the first draft of the specification simply by providing natural language prompts. Furthermore, LLMs can also be employed to generate more complex code compared to traditional code generators and according to the desired conventions, thus minimizing human error. However, one of the limitation of current generation of LLMs is that they can produce only few hundred lines of code at a time [23]. Fortunately, this limitation of LLMs aligns very well with microservice architecture because each service is independent, small and focuses on particular problem of the large system. Hence integrating LLMs can significantly boost developer's productivity.

This thesis presents a system that accelerates the generation of OpenAPI specifications and API code. By integrating a chat interface, it allows developers to refine the generated code through natural language prompts, easing the development process from design to fixing. The generated code will follow predefined folder organization and also deployable in the Docker environment. A key feature of this system is its ability to access the logs from local development environment in order to guide and assist developers in debugging the service code. The system consolidates all these tasks under a single interface, significantly reducing the need for switching between multiple tools.

To validate the system's functionality and usability, we conducted a survey with industry practitioners who had varying levels of experience in software development and microservice architecture. This allowed us to capture a broad spectrum of feedback on the system's usability and effectiveness across different expertise levels. Additionally, we made the data from our data analysis publicly available, providing transparency and enabling further insights from the development community [28].

2 Background

2.1 Generative AI

Generative Artificial Intelligence (AI) is a branch of machine learning that can create realistic and complex data, like text or images, by learning patterns and structures from existing data[13]. A wide range of fields such as technology, business, education, healthcare, and arts have been affected directly or indirectly by Generative AI [8][6]. Even though it introduces some challenges like mode collapse, evaluation difficulties, ethical issues, and data quality but it also offers diverse opportunities for amplifying creativity and productivity. There is a need for proper AI-human alliance so that these challenges can be eliminated and benefits can be maximized[19][12][13].

Generative AI presents several technical challenges. One of the most common ones is hallucination, where it produces content that often looks correct but is factually incorrect or fabricated[2]. Depending on the industry, this can lead to serious consequences [26]. Incorporating Human-in-the-Loop (HITL) approaches can address concerns related to the accuracy of AI-generated content. By involving humans in tasks where precision is critical, the risk of hallucinations or incorrect outputs in the final product can be significantly reduced. Additionally, human feedback can provide rapid evaluations of generated content, helping the model to refine and improve the accuracy of its outputs over time.

2.2 LLMs for Code Generation

Large Language Models in simple terms are machine learning models that are trained using vast datasets giving them the ability to understand structures and patterns in data across different domains [34] [33]. In recent years, LLMs have become a very popular choice for code generation tasks. These models utilizes both natural language understanding and generative capabilities, and have shown remarkable performance in code synthesis[17]. This has attracted the attention of many academic researchers and software developers [11]. Another application of these code generating model is code completion where code snippets are suggested based on partially written code. Despite many benefits and capabilities, LLMs in code generation tasks face some challenges. One of the most common and concerning ones is the quality of code snippet generated, which often contains bugs or security vulnerabilities. As mentioned earlier, human involvement in the process of using LLMs to solve a particular problem becomes very crucial. Furthermore, fine-tuning LLMs on datasets containing vulnerability fixes can also address the security concerns in generated code [32].

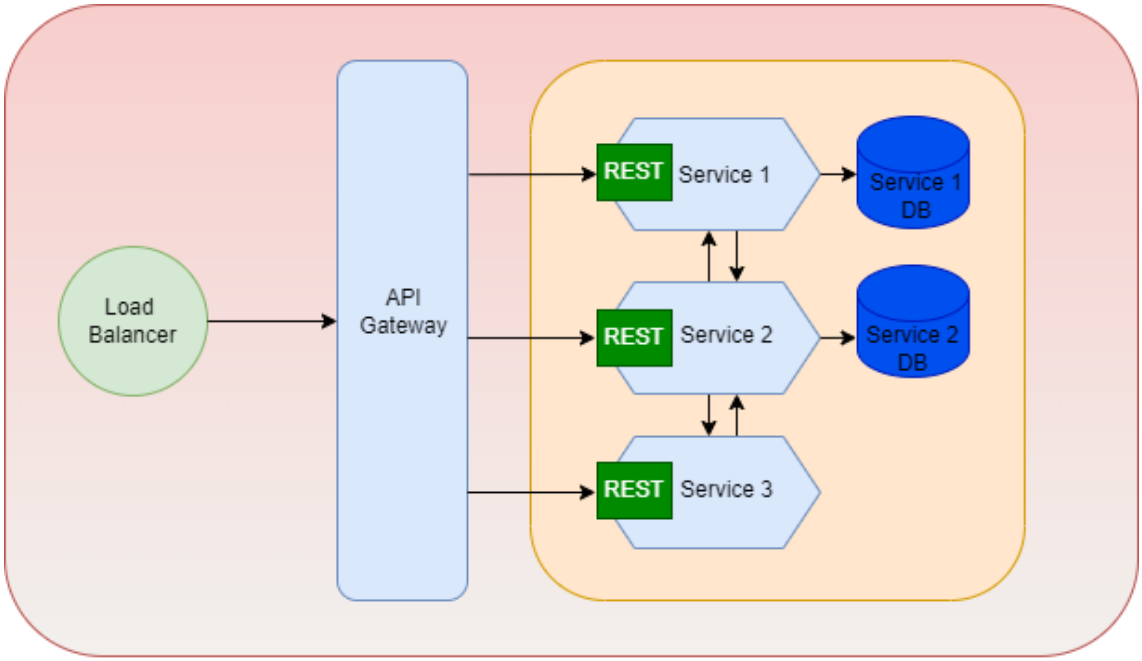

Figure 2.1 *Microservice Architecture*

2.3 Microservices Architecture

This section provides an overview of Microservices Architecture, a modern software design pattern that breaks down applications into smaller, independent services [16], [21], [22]. It will cover the key concepts, advantages, and challenges of using microservices in comparison to traditional monolithic architectures.

2.3.1 Key Characteristics of Microservices

Here are some of the key characteristics that define microservices architecture. These traits make microservices distinct from traditional monolithic designs and contribute to their scalability, flexibility, and modularity [3], [10].

Independence: Each microservice operates as an independent unit. This independence enables teams to develop, test, deploy, and scale services individually, reducing the risk of system-wide failures [10].

Loosely Coupled: Microservices interact with each other through well-defined APIs but are not tightly dependent on each other. This ensures that changes made to one microservice have minimal impact on others[31].

Single Responsibility: Each microservice is designed to perform a specific task or function. This makes the system more modular and easier to understand,

develop, and maintain [3].

Technology Agnostic: Microservices can be built using different programming languages, frameworks, and databases, offering greater flexibility and allowing teams to choose the best tools for each microservice.

2.3.2 Advantages of Microservices

Microservices architecture offers several benefits that address the limitations of traditional monolithic designs. These advantages make it the preferred choice for developing scalable, resilient, and maintainable applications. Below, are some key advantages of adopting microservices [5].

Scalability: Microservices enable independent scaling of services. Each service can be scaled up or down based on its specific demand, optimizing resource utilization and reducing operational costs.

Flexibility: Teams can develop different microservices using diverse programming languages, frameworks, or databases. This technology-agnostic approach allows the use of the best tools suited for each service, enhancing innovation and efficiency.

Faster Development and Deployment: Since microservices are developed and deployed independently, teams can release updates or new features more frequently. This accelerates the development lifecycle and reduces the time-to-market for new functionalities.

Improved Fault Isolation: The modular nature of microservices ensures that a failure in one service does not cascade across the entire system. This enhances the overall reliability and robustness of the application.

Reusability: Microservices designed for specific functionalities can be reused in other applications. This reduces development effort and supports faster creation of new solutions.

Continuous Integration and Deployment (CI/CD) Friendly: Microservices align well with modern DevOps practices. Their decoupled architecture allows for seamless integration with CI/CD pipelines, enabling automated testing and deployment processes.

Enhanced Maintenance: The smaller and modular codebase of each microservice simplifies debugging and maintenance, reducing the time and effort required during the maintenance phase of the software lifecycle.

2.3.3 The Need for Microservices

While monolithic architectures are suitable for simple applications and provide a clear understanding of the codebase during initial development, they pose significant challenges as systems grow. As applications scale and user requirements become more complex, monolithic structures struggle to keep pace with evolving needs. Below are some of the primary limitations of monolithic architectures that highlight the need for microservices [5].

Scalability Challenges: Monolithic applications become difficult to scale as user demands increase, leading to performance bottlenecks. Scaling a monolithic system often requires duplicating the entire application, even when only a specific component needs additional resources. This inefficiency can result in higher operational costs and wasted resources [3].

Maintenance Overhead: The maintenance phase in a monolithic structure is time-consuming due to the tightly coupled nature of the codebase. A single change in one part of the system can require extensive testing and redeployment of the entire application. This slows down development cycles and increases the risk of introducing bugs or breaking existing features.

Technology Constraints: Monolithic architectures often lock development teams into a single technology stack, limiting the ability to integrate newer, more efficient tools. Adopting innovations or transitioning to modern technologies becomes challenging, as it might require significant rewrites of the entire application.

Limited Agility: Monolithic systems typically follow a rigid structure, making it difficult to implement changes or add features quickly. This lack of flexibility hampers an organization's ability to adapt to changing business needs and market demands.

Deployment Challenges: Continuous integration and deployment are harder to achieve in monolithic architectures. Deploying updates requires rebuilding and redeploying the entire application, increasing downtime and complicating the deployment process.

Risk of System-Wide Failures: Since monolithic applications are tightly integrated, a failure in one part of the system can propagate, potentially causing the entire application to fail. This lack of fault isolation can severely impact reliability and availability [29].

In contrast, microservices offer a modular and distributed approach that addresses these challenges effectively. By breaking down applications into smaller, independent services, microservices enable [5]:

- **Targeted Scaling**: Individual services can be scaled independently based on demand, optimizing resource usage and improving system performance.
- **Faster Development Cycles**: Teams can work on different services simultaneously, accelerating feature delivery and reducing time-to-market.
- **Resilience**: Faults are isolated to specific services, ensuring the rest of the system remains operational even during failures.
- **Technological Freedom**: Teams can choose the best tools, frameworks, and programming languages for each service, fostering innovation and efficiency.
- **Seamless Deployment**: Continuous integration and deployment pipelines are easier to implement, supporting frequent updates with minimal disruption.

Microservices thus empower organizations to build systems that are scalable, maintainable, and adaptive to evolving business and technological requirements.

2.3.4 Challenges with Microservices

While microservices offer numerous advantages, adopting this architecture comes with its own set of challenges. Developing, deploying, and managing a distributed system of independent services introduces complexities that require careful consideration and robust strategies. Below, we discuss some of the key challenges associated with microservices [27].

Increased Complexity: Unlike monolithic architectures, microservices require managing a distributed system. Each service operates independently, which means developers must account for service discovery, communication protocols, and dependency management. The increased number of components can make debugging and monitoring more complex.

Communication Overhead: Microservices rely on APIs for inter-service communication. This introduces latency and the potential for network failures. Ensuring secure, efficient, and reliable communication between services becomes a critical challenge, particularly in high-load environments.

Data Management: Unlike monolithic systems that use a single database, microservices often involve distributed data management. Maintaining consistency across multiple databases, implementing data replication, and managing transactions across services can be difficult and resource-intensive.

Deployment and Orchestration: While microservices enable independent deployment, managing the deployment of numerous services can become complex. Tools like Kubernetes and Docker are often necessary for container orchestration and automation, requiring teams to learn and implement additional technologies.

Monitoring and Debugging: The distributed nature of microservices makes monitoring system performance and debugging issues challenging. Observability tools must be implemented to provide detailed insights into service interactions, error rates, and latency. Logging, tracing, and metrics collection become critical but can add overhead.

Testing Challenges: Testing a microservices-based system is more complex than testing a monolithic application. Integration testing requires simulating multiple service interactions, and maintaining test environments for distributed systems can be costly and time-consuming.

Service Dependency Management: Ensuring that all services work seamlessly together requires robust dependency management. Versioning APIs and managing backward compatibility are necessary to prevent breaking changes that can disrupt the system.

Security Concerns: Microservices increase the attack surface compared to monolithic systems. Each service must be secured independently, requiring careful implementation of authentication, authorization, and encryption mechanisms. Secure API gateways are critical to manage and protect communication between services.

Team Coordination: The autonomy of microservices development can lead to challenges in coordination among teams. Ensuring consistency in design, documen-

tation, and development practices requires effective communication and governance.

Despite these challenges, the benefits of microservices often outweigh the drawbacks, particularly for large-scale, dynamic systems. Organizations adopting microservices must invest in robust infrastructure, skilled teams, and modern tools to overcome these challenges and maximize the potential of this architecture [29].

2.4 API-First Approach

This section covers the API-first approach, a design philosophy that emphasizes defining and designing application programming interfaces (APIs) before implementing the logic of services. It prioritizes API design as a foundational step, so that the system is modular, collaborative, and scalable. By treating APIs as first-class citizens, the API-first methodology addresses challenges in distributed systems and speeds up the development cycles, especially in microservices architectures.

2.4.1 Overview of API-First Approach

The API-first approach is a design philosophy that emphasizes the importance of defining APIs before initiating service implementation. This methodology ensures that APIs act as the central point of interaction between different services, fostering a unified and modular system design. By prioritizing the definition of inputs, outputs, and communication protocols upfront, the approach addresses potential integration issues early in the development process.

Key Benefits of Early API Definition: Defining APIs at the beginning of the development cycle offers several benefits:

- **Primary Interaction Medium:** APIs serve as the primary channel of communication between services, ensuring seamless data flow and reducing interdependencies.
- **Shared Understanding:** Teams across different functions, such as developers, testers, and product managers, gain a clear understanding of API behaviors and use cases. This shared understanding minimizes ambiguities and accelerates alignment.
- **Promotes Modularity:** The focus on clear, well-documented API contracts ensures that services remain modular and independent. This modularity enhances maintainability, as individual components can be updated or replaced without affecting the overall system.

Encouraging Collaboration and Independence: The API-first philosophy prioritizes collaboration by defining clear contracts that serve as a single source of truth. This enables teams to work in parallel, allowing API consumers and providers to develop independently while adhering to agreed-upon standards. Furthermore, this independence accelerates development and facilitates the scalability of individual services.

Addressing Integration Challenges: By establishing predefined API standards, the approach mitigates the challenges often encountered during the integration phase. Developers and teams can rely on consistent communication protocols and behavior expectations, making it easier to integrate services, even when they are developed using different technologies or by separate teams.

In summary, the API-first approach is a proactive strategy that sets the stage for efficient, scalable, and collaborative development processes, particularly in systems that rely on distributed architectures like microservices.

2.4.2 Role in Microservices Architecture

In microservices architectures, the API-first approach plays a critical role in structuring and scaling systems. Its implementation includes:

- Breaking down applications into smaller, independent services.
- Defining API contracts for these services before development begins.
- Sharing interface specifications across teams to ensure alignment between service providers and consumers.
- Leveraging suitable tools for synchronous or asynchronous communication.

This structured process allows microservices to integrate seamlessly, enabling flexibility and scalability [9].

2.4.3 Standardization with OpenAPI Specification

To ensure consistency and clarity, the API-first approach often relies on standards like the OpenAPI Specification (OAS) [20]. Figure 2.2 shows an example of such a specification for a User service. Key advantages of OAS include:

- **Clarity**: OAS provides a well-defined structure for documenting REST APIs.
- **Interactive Documentation**: Automatically generated documentation helps teams understand and test API behavior.

- **Version Control**: API changes can be tracked, ensuring effective communication across teams.
- **Code Generation**: OAS supports the generation of client and server stubs, reducing development effort [30].

By adopting OAS, teams establish a single source of truth for API design, fostering better collaboration and integration.

2.4.4 Parallel Development and Testing

One of the key strengths of the API-first approach is its ability to support parallel workflows. This includes:

- **Concurrent Development**: Client and server teams can work simultaneously, as the API contracts serve as the definitive reference.
- **Efficient Testing**: Test cases can be planned and executed earlier, allowing quicker validation of service implementations.
- **Early Fault Detection**: Synchronous development ensures that design flaws are identified and addressed promptly.

This enables faster feature delivery and aligns well with agile development practices [7].

2.4.5 Benefits of API-First Approach

The API-first approach offers several advantages, particularly for microservices architectures:

- Enhanced modularity and reusability of services.
- Faster development cycles through parallel workflows.
- Improved collaboration among development, testing, and deployment teams.
- Clear documentation and version control for better maintainability.

By prioritizing API design, this approach ensures scalability, reliability, and efficiency in building modern distributed systems .

```
{
  "openapi": "3.0.3",
  "paths": {
    "/users": {
      "get": {
        "summary": "Get all users",
        "responses": {
          "200": {
            "description": "List of users"
          }
        }
      },
      "post": {
        "summary": "Create a user",
        "responses": {
          "201": {
            "description": "User created"
          }
        }
      }
    },
    "/users/{id}": {
      "get": {
        "summary": "Get user by ID",
        "responses": {
          "200": {
            "description": "User details"
          }
        }
      },
      "put": {
        "summary": "Update user by ID",
        "responses": {
          "200": {
            "description": "User updated"
          }
        }
      },
      "delete": {
        "summary": "Delete user by ID",
        "responses": {
          "204": {
            "description": "User deleted"
          }
        }
      }
    }
  },
}
```

Figure 2.2 *OpenAPI Specification for the User Service*

3 Research Methodology

In this section, we present the methodology for automating the entire development of a service from generating Open API specification to testing and fixing the API code. Section 3.1 provides details of the formulated Research Questions (RQs). The system design and multi-agent workflow are discussed in Section 3.2 and we discuss the details of our evaluation framework in Section 3.4.

3.1 Research Questions (RQs)

Based on our study goal, we formulated the following two Research Questions (RQs).

> ***RQ1.*** *How do users perceive the usability and effectiveness of a system that automates code generation and testing, compared to traditional manual coding methods?*

The main objective of **RQ1.** is to assess user perceptions of the usability and effectiveness of an AI-driven system for code generation and refinement. This includes comparing the AI system with traditional manual coding methods regarding ease of use, efficiency, and user satisfaction.

> ***RQ2.*** *To what extent does the system reduce the need for switching between multiple development tools in the development and testing of microservices?*

The main objective of **RQ2.** is to evaluate how effectively the system minimizes the need for switching between different development tools. This involves determining how much the AI system simplifies the process by integrating multiple functions, such as coding, testing, and deployment.

> ***RQ3.****What impact does the system have on reducing manual coding efforts and increasing the speed of microservice development?*

The main objective of **RQ3.** is to analyze how the system reduces manual coding efforts and accelerates microservice development.

3.2 System Design

The system is scoped to the development of services that perform Create, Read, Update, and Delete (CRUD) operations and will communicate using REST architectural style, which is a perfect fit for these services because of its ability to handle

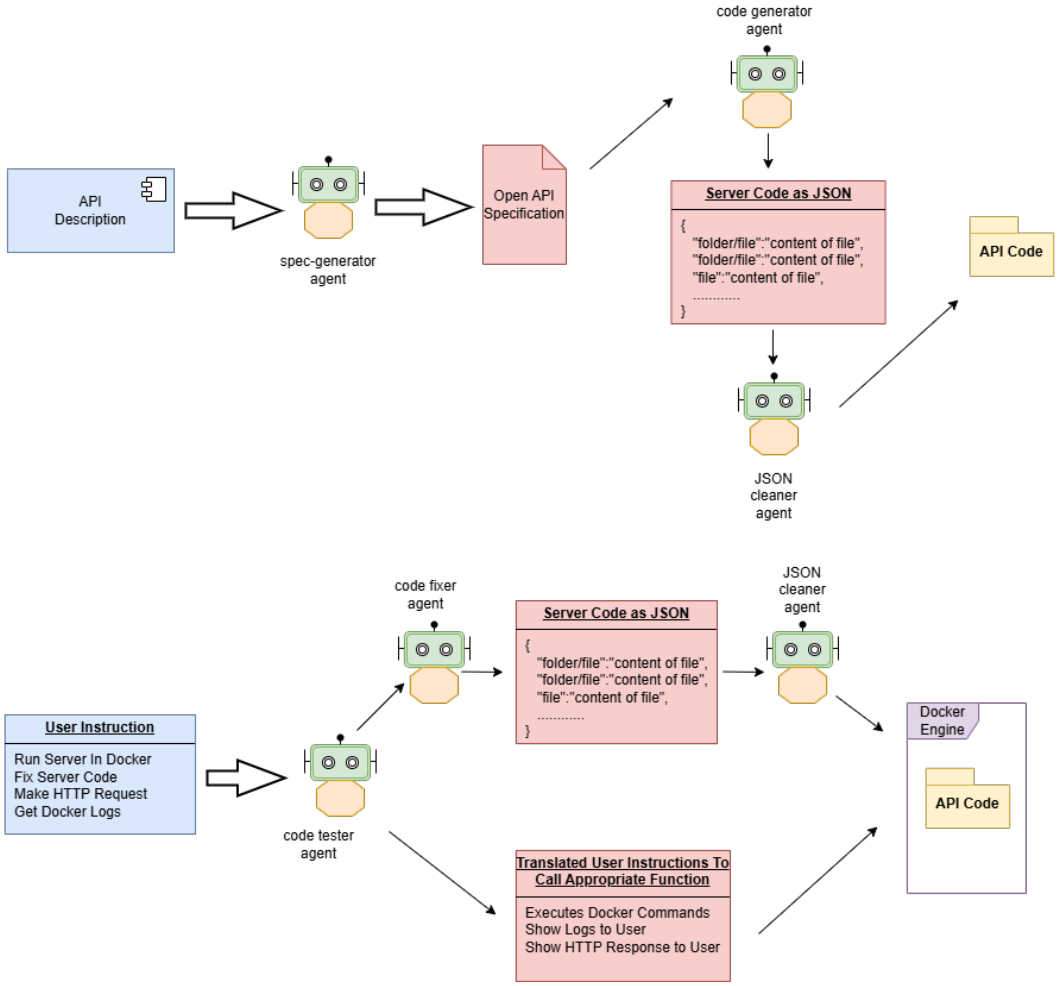

Figure 3.1 *System Design*

CRUD operations using HTTP methods such as POST, GET, PUT and DELETE [1]. To define the API specification the system will use OpenAPI Specification standards as it provides a complete framework for defining REST APIs [4]. The LLM that is responsible for interacting with user is GPT-4 by OpenAI because of its great performance in generating accurate code snippets and interpreting complex natural language prompts. Moreover, its advanced function calling feature makes it a suitable model for this type of system [35].

The system's workflow is quite structured which starts by taking input from the user and transitions into testing of generated server code. For this task, we have used a multi-agent framework where each LLM agent has a distinct role. Details about the agents can be found in Table 3.1 which also contains details of functions that a particular agent can call in order to interact with the local environment of the user. Following are the steps involved in the process of generating, validating, and fixing the code for a service:

3.2.1 OpenAPI Specification Generation

The process starts with the user specifying high-level requirements for the service. These requirements typically contains details such as the structure of data models, the desired endpoints for communication, and the types of database operations that need to be implemented. Using these inputs, the system utilizes the GPT-4 model to generate an initial draft of the OpenAPI Specification (OAS). This task is primarily undertaken by the spec-generator agent, a core component of the system.

The creation of this initial OpenAPI Specification is a critical step in the workflow, as it defines the service interface in a precise and standardized manner. By concretely defining the service interface, the system sets a solid foundation for further development stages, minimizing ambiguities and potential errors. This detailed approach facilitates a structured development process, helping in both the implementation of the service and its subsequent validation and refinement phases.

3.2.2 Finalization of specification

After the initial OpenAPI specification is generated, the user is provided with the opportunity to review the output. At this stage, the user can examine the specification for accuracy and completeness, making any adjustments or providing additional details to the spec-generator agent. This feedback loop allows for a dynamic, iterative refinement of the specification, ensuring that it aligns closely with the user's specific needs and expectations. Each iteration is aimed at enhancing the specification's precision, clarity, and relevance, while also ensuring compliance with the OpenAPI Specification (OAS) standards.

As the specification progresses, the user has the flexibility to suggest further modifications, whether in terms of adding or removing endpoints, adjusting the data models, or specifying certain behaviors of the service. This phase is critical for achieving a high-quality and user-centric specification. The process continues until the specification accurately reflects the service's requirements, with all necessary elements in place for further development.

Once the user is satisfied with the final output, the user can command the agent to save the specification to the local environment. Leveraging GPT-4's function-calling capabilities, the agent triggers the appropriate function to securely store the specification file. This finalized version of the specification serves as a comprehensive blueprint or contract for the service's development, acting as the foundation upon which the entire service architecture will be built. By establishing a clear, well-defined starting point, the specification ensures consistency and correctness throughout the development lifecycle, allowing all subsequent development steps to proceed smoothly and efficiently.

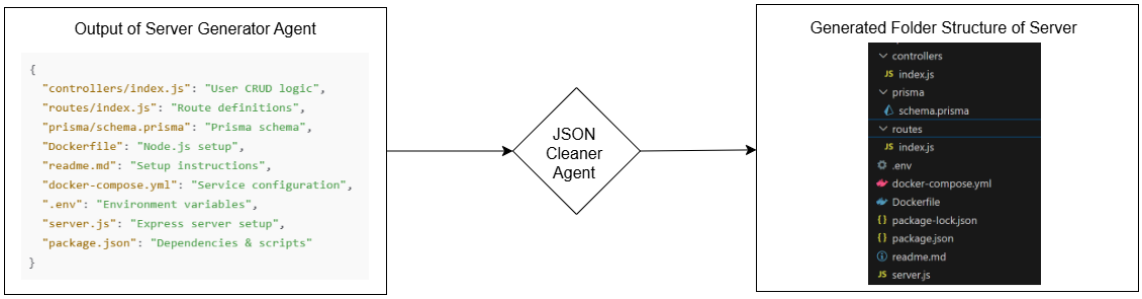

Figure 3.2 *JSON to directories*

This approach not only saves time but also significantly reduces the potential for errors during development, fostering a higher level of code quality and overall service integrity. Moreover, the clear contract created by the specification will act as a reference point for the developers and help maintain alignment between stakeholders as the project progresses. The outcome is a more reliable and scalable service, driven by a well-structured and agreed-upon specification.

3.2.3 Server Code Generation

Once the OpenAPI specification is finalized, it is provided to the Server Code Generator, which contains two primary agents: the code-generator and the JSON-cleaner. The role of the code-generator is to take the specification and output a structured JSON string that represents the desired folder structure and corresponding file contents for the server code. The JSON format is designed in a way where the key denotes the directory or file path (e.g., ”server/index.js”), and the value holds the content of that file (e.g., the code for ”index.js”) as shown in Figure 3.2. This structure ensures that the generated server code is logically organized, with directories and files properly named and filled with the appropriate content.

However, the JSON string produced by the code-generator may sometimes contain invalid tokens, which could lead to parsing errors when attempting to process the JSON. To mitigate this issue, the string is passed to the JSON-cleaner agent, whose job is to clean the string and resolve any issues that might cause errors during the parsing process. The JSON-cleaner ensures that the data is in the correct format and free from any issues that could disrupt the subsequent steps.

Once the JSON string is cleaned and properly formatted, the JSON-cleaner agent then calls a function that parses the JSON and translates it into actual directories and files in the user’s local environment. The function creates the necessary directory structure based on the keys in the JSON (representing file paths), and it fills each file with the corresponding content (the values in the JSON). This step effectively generates the server code, organizing it in a logical and structured manner, with each file placed in the appropriate directory.

The final output of this process is the server code saved in the user's local environment, now fully structured and ready for further development. The code is organized according to the specifications of the OpenAPI blueprint, ensuring consistency, clarity, and ease of navigation for the development team. By automating this step, the system reduces the risk of human error and accelerates the overall development process, as developers can focus more on writing business logic and functionality rather than manually creating and organizing the project files.

3.2.4 Automated Validation and Execution

To interact with the server code, the system employs a code-tester agent, which plays a pivotal role in validating and testing the functionality of the generated server code. This agent heavily leverages the function calling feature of GPT-4, allowing it to automate complex tasks that are typically handled manually by developers.

One of the core functions that the code-tester agent calls is the docker-compose up command. This command is crucial for building and loading the Docker container in the user's local environment. When executed, it sets up the entire environment for the server code to run within a container, ensuring consistency across different machines and facilitating the testing process by mimicking real-world deployment scenarios.

In addition to managing the Docker container, the code-tester agent can call other functions to retrieve critical information, such as logs of the container and the status of services running within the container. These logs and status data are often verbose and difficult to read directly from the console. To address this, the agent processes and organizes this information into a more user-friendly format that can be easily understood, allowing the user to quickly assess what is happening in the Docker engine without sifting through raw log files.

Furthermore, the code-tester agent enhances the testing experience by enabling users to interact with the server through natural language prompts. Instead of requiring the user to switch between multiple tools, such as Postman or curl, to make requests and validate the server's functionality, the agent allows users to directly send API requests and view the responses within the same interface. This integration simplifies the testing process, reducing the need for external tools and offering a more cohesive and efficient workflow for validating the server code's behavior.

By using GPT-4's function calling capability, the code-tester agent automates complex testing processes, making it easier for users to deploy, monitor, and validate their server applications. This seamless interaction not only improves developer productivity but also ensures that the server functions as expected, without the need for additional manual intervention.

Agent	Job	Functions Calls	Function Description
spec-generator	Generate OpenAPI specification from user API description	save_openapi_spec	Saves the given OpenAPI specification text to a YAML file and returns success/error.
code-generator	Generate code for server in JSON format and save it in user's working directory	save_json	Validates and fixes a given JSON object before saving it as server code.
json-cleaner	Cleans JSON data of server files so that it can be parsed without error		
code-fixer	Takes server code in JSON format with instructions for making fixes or updates and updates the code	save_json	Validates and fixes a given JSON object before saving it as server code.
code-tester	Executes docker commands to start containers and fetches logs from containers. Also, send requests to service and show results to users. It is also responsible for making changes to the server code	run_docker_compose docker_compose_status get_docker_compose_logs run_curl_command update_json	start the services in local docker engine get status of container related to docker compose file get logs of containers used to send http requests to services update/fix server code based

Table 3.1 *LLM Agent Details*

3.2.5 Iterative Code Fixing

After the user gains the ability to interact with the server code—such as starting and running containers, making requests to server containers, and retrieving logs of running containers—there might be scenarios where the server does not function as expected or the user encounters issues. In these situations, the code-tester agent plays a crucial role by analyzing the logs it collects from the containers to help diagnose potential problems. Based on the insights it gathers, the code-tester agent can suggest possible fixes to the user.

In instances where the suggested fix is not immediately obvious or the user prefers an automated approach, the code-tester agent can further interact with the code-fixer agent. This collaboration allows the system to automatically apply fixes by modifying the already saved server code. This step eliminates the need for users to manually inspect the code and make updates themselves, which can be time-consuming and error-prone. By automating this process, the system significantly boosts productivity and reduces the time spent troubleshooting and fixing issues.

The process begins when the code-tester agent identifies a potential fix based on the logs and communicates this to the code-fixer agent. The code-fixer agent then takes two inputs: the current version of the server code and the identified fix. These inputs are provided by the code-tester agent, ensuring that the fix is relevant to the encountered problem. The code-fixer agent processes these inputs and generates an updated version of the server code, represented as a JSON string. In this string, each directory is a key, and the corresponding content of the file is the value.

Once the code-fixer agent outputs the updated code in JSON format, the system parses this JSON string and saves the changes, which automatically updates the server code in the user's local environment. This step ensures that the fix is applied to the server code, making it consistent with the required adjustments. After the server code is updated, the system issues a command to Docker to rebuild and restart all the affected services, ensuring that the latest version of the code is running in the container.

This process of identifying, suggesting, and applying fixes is repeated as necessary, with the user remaining involved in the loop to ensure the desired outcome. The system continuously refines the code based on user's input, until the issue is resolved and the server operates as expected. By automating the process of code fixing and server updates, the system dramatically streamlines the development and testing workflow, saving valuable time.

3.2.6 Generated Server Code

The generated code represents a modern application architecture using several widely adopted technologies to create a scalable, efficient, and maintainable service. This section discusses the technologies that are utilized to structure the application, handle server-side logic, manage database interactions, and deploy the system in a containerized environment.

The generated server code serves as the foundation for a RESTful API, providing the necessary endpoints for interaction with the database and handling HTTP requests. The code is built with **Node.js** and **Express.js**, which work together to create a non-blocking, efficient server capable of handling concurrent requests. Express.js manages routing, middleware, and HTTP responses, while Node.js ensures the asynchronous processing of requests, improving scalability.

The code follows a modular structure, with separate directories for controllers, routes, and Prisma database models. The `controllers/` directory houses the business logic for the API, while the `routes/` directory defines API endpoints, ensuring clear separation of concerns. The Prisma `schema.prisma` file models the database schema and handles all database interactions.

The generated server code also integrates with **Prisma**, an ORM that simplifies database queries and migrations. Prisma abstracts away raw SQL queries and provides a type-safe API for interacting with PostgreSQL, reducing the complexity of working with a relational database.

The server code is containerized using **Docker**, allowing the entire application to run in isolated containers. Docker Compose is used to manage the containers, ensuring that both the application server and the PostgreSQL database can run in a unified environment, with their interactions defined in a `docker-compose.yml` file.

Node.js & Express.js (Backend Framework)

The backend of the application is built with **Node.js**, an event-driven, non-blocking I/O model that is ideal for building scalable applications. Coupled with **Express.js**, a minimal and flexible web application framework, the system is able to handle HTTP requests and responses efficiently. Express.js simplifies routing and middleware management, which makes it easy to organize and structure the application.

Prisma (ORM for Database Interaction)

Prisma is an advanced **Object-Relational Mapping (ORM)** tool used to interact with the database. It simplifies complex database queries by generating type-safe queries based on the schema defined in the `schema.prisma` file. Prisma allows

developers to work with the database in an intuitive way, improving developer productivity and reducing the risk of SQL injection. It also provides advanced features like migrations and database seeding, which streamline database management.

PostgreSQL (Database)

PostgreSQL is the chosen relational database management system for this application. Known for its stability, scalability, and full compliance with ACID principles, PostgreSQL is well-suited for handling complex data relationships and large datasets. It offers robust features for data integrity and performance, making it an ideal choice for production-grade applications.

Docker & Docker Compose (Containerization)

Docker and **Docker Compose** are used to containerize the application, ensuring a consistent and reproducible environment for development and production. Docker allows the server and database to be packaged into separate containers, encapsulating their dependencies and configurations. Docker Compose simplifies the management of multi-container setups by providing an easy-to-use YAML configuration file that defines how the containers interact with each other.

Environment Variables (.env)

The application makes use of **environment variables**, stored in a `.env` file, to manage sensitive data and environment-specific configurations, such as database credentials and server ports. This ensures that no sensitive data is hardcoded into the source code, enhancing security and allowing the same codebase to be used across different environments (development, staging, and production).

File Structure and Modularity

The application is organized with a modular file structure that adheres to best practices. The `controllers/` directory handles business logic, while the `routes/` directory defines the API endpoints. The `prisma/` directory contains the schema file that models the database, and the `server.js` file is responsible for setting up and configuring the Express server. This structure promotes scalability and maintainability by separating concerns and making it easier to add new features.

Readme.md (Documentation)

A `README.md` file provides essential documentation about the setup and usage of the application. It includes instructions for installing dependencies, running the

application, and configuring environment variables. This documentation is valuable for developers and contributors who want to understand the application's structure and get up and running quickly.

Conclusion

The combination of **Node.js**, **Express.js**, **Prisma**, **PostgreSQL**, **Docker**, and **environment variables** creates a robust and modern web application stack. By leveraging these technologies, the application achieves flexibility, scalability, and maintainability. The use of Docker ensures that the system can be easily deployed and scaled, while Prisma simplifies database interactions and enhances developer productivity. This stack is well-suited for building modern, high-performance web applications.

3.3 Implementation

This section highlights the methodology employed to build a structured and user-focused web application. The development process was focused on the creation of robust backend architecture and an intuitive user interface. By using modern tools and frameworks, the system ensures that the application is reliable, maintainable, and capable of delivering a seamless user experience. The use of modern frameworks and design principles allows the application to adapt to evolving user needs and technological advancements while maintaining robust performance and user satisfaction.

3.3.1 Front-end

The front-end of the application was developed using the React library, leveraging its component-based architecture to build a highly dynamic and responsive user interface. React's declarative approach simplifies the process of creating complex UIs, ensuring that the application remains maintainable and easy to scale over time. To further optimize the development workflow, the Vite build tool was used, providing a fast and efficient development environment with hot module replacement (HMR) for real-time feedback during the development process.

The combination of React and Vite enabled the team to rapidly prototype and iterate on the user interface, leading to the creation of a seamless and intuitive experience. Vite's fast build times and support for modern JavaScript features, such as ES modules, made the development process significantly more efficient. React's ecosystem of libraries, along with the flexibility of Vite, allowed the application to integrate various features, such as state management, routing, and form handling, without sacrificing performance.

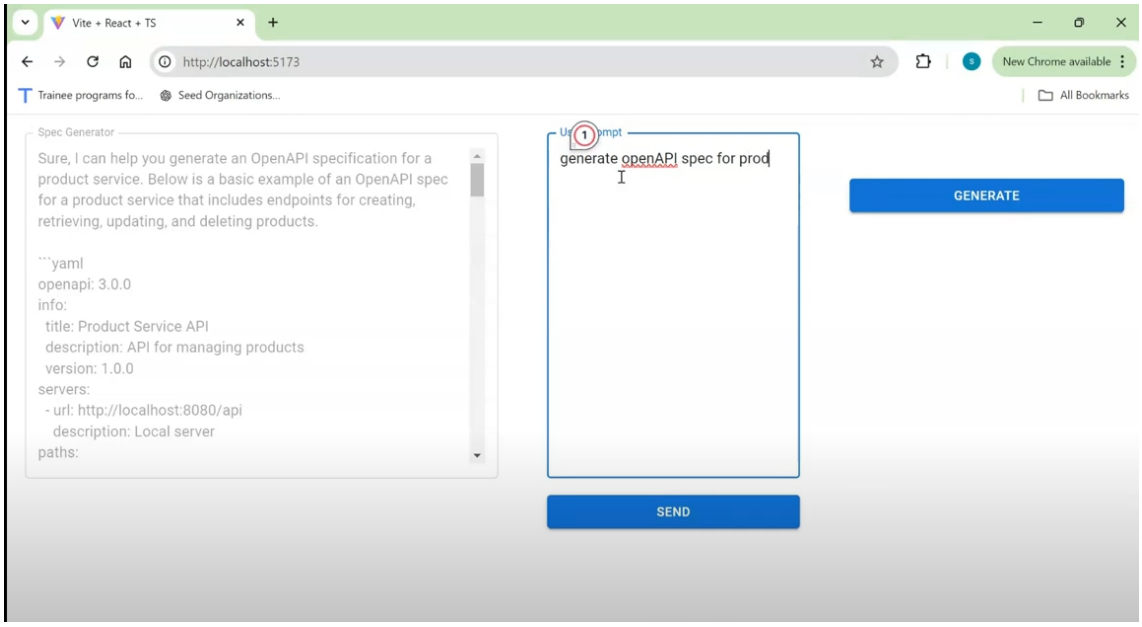

Figure 3.3 *Specification Generation Screen*

The front-end was designed with a focus on user experience, ensuring that it adapts to various screen sizes and devices through responsive design principles. By utilizing reusable components and a modular approach, the codebase remains clean and easily maintainable, making future updates and enhancements more manageable. This front-end architecture ensures that the application delivers a seamless, fast, and user-friendly experience across a range of devices and user needs.

Specification and Server Generator Screen

3.3.2 Specification and Server Generator Screen The Specification and Server Generator Screen is designed to streamline the process of generating and refining OpenAPI specifications and server code. It contains an input box where users can enter prompts related to the OpenAPI specification. Once the user inputs their specification prompt, the system processes it and displays the generated specification in another disabled input field. This allows users to review the generated specification and make sure it meets their requirements. If the user is satisfied with the generated specification, they can proceed to request the generation of the server code. This screen ensures an efficient and user-friendly interface for creating and refining specifications while maintaining flexibility for user customization.

Server Interaction Screen

The Server Interaction Screen provides users with an interface to interact with the server directly. It features an input box and a button, where users can provide commands to control the server. Commands can include running the server, retrieving

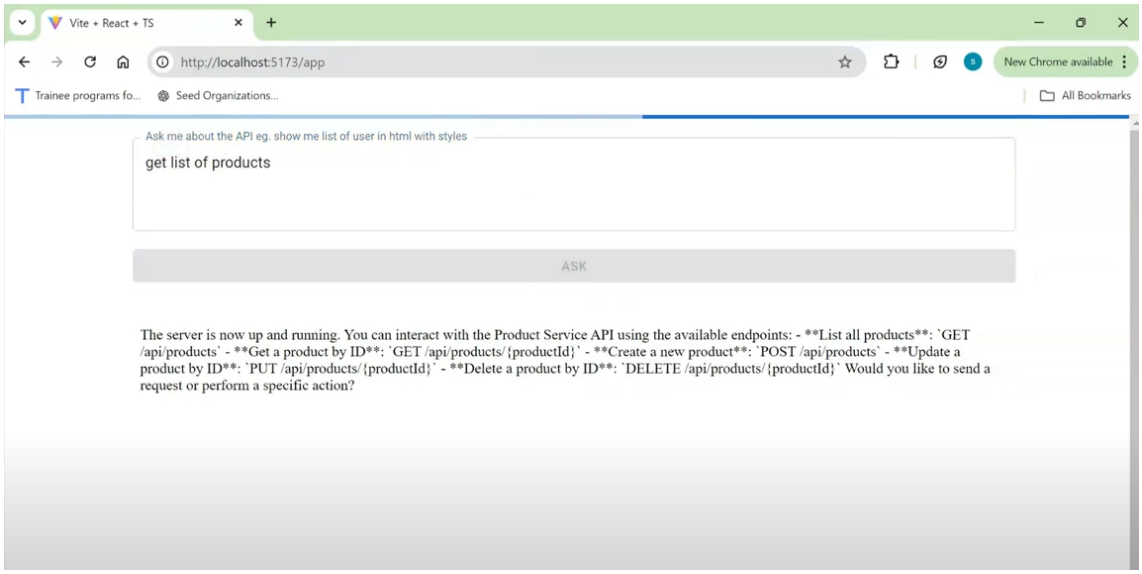

Figure 3.4 *Server Interaction Screen*

server logs, or requesting the system to fix any issues with the server code. Once a command is provided, the system executes the job and displays the output below the button, offering real-time feedback on the status of the command or job. This interaction model allows users to easily monitor and control the server, improving workflow efficiency and ensuring that any issues can be quickly addressed.

3.3.2 Back-end

The backend of the application is built using Flask, a lightweight and flexible web framework for Python. Flask facilitates the development of the server-side logic that powers the application, handling HTTP requests, processing data, and coordinating interactions between the front-end and the various services integrated into the system.

The backend serves as the central hub for executing critical operations such as generating OpenAPI specifications, creating server code, and enabling interactions with the server through command execution. It acts as an intermediary between the user interface and the various external tools used in the application, ensuring that the user's requests are processed accurately and efficiently.

Flask is chosen for its simplicity and ease of use, allowing for rapid development and straightforward integration of additional functionality as needed. The application is designed to handle multiple types of user requests, which are processed by routes defined within the Flask app. To ensure that the backend can support interactions from different origins, CORS (Cross-Origin Resource Sharing) is enabled.

Furthermore, the backend makes use of external libraries and services to enhance its capabilities. For instance, the OpenAI API is utilized to provide natural lan-

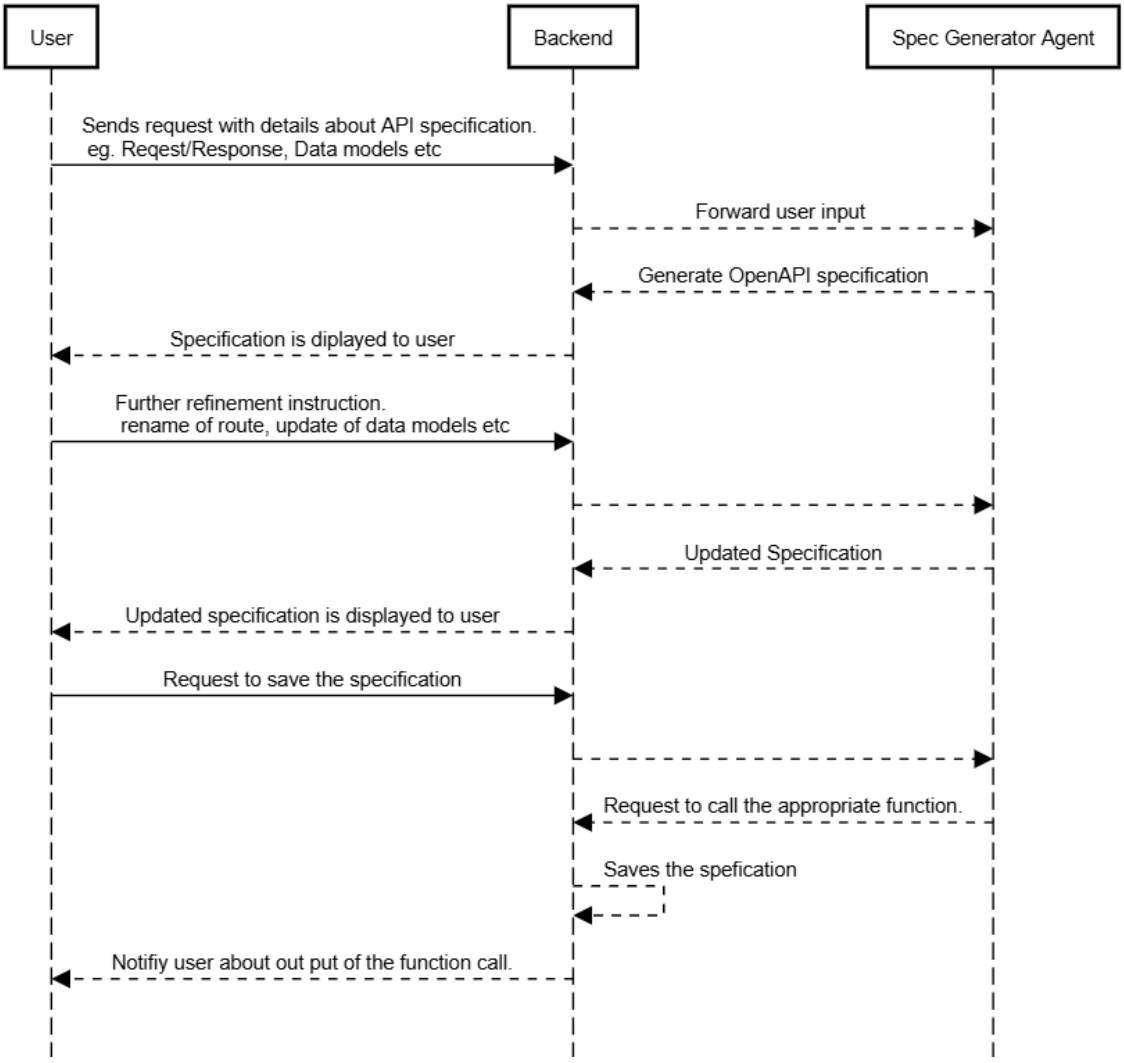

Figure 3.5 *Specification Generation*

guage processing features, enabling the application to interact intelligently with the user and perform tasks like server code fixing and querying server status. The backend also integrates various command-line tools to interact with Docker containers, manage API requests, and fix server code based on instructions provided by the user.

The backend is designed to be robust, secure, and scalable, ensuring that it can handle a variety of user interactions while maintaining performance and responsiveness. Below are the key components of the backend

Spec Generation (/spec Route)

The `/spec` route is a key feature of the backend, designed to handle the generation of OpenAPI specifications based on user-provided inputs. Users interact with this route by supplying a message or prompt that describes their desired API structure.

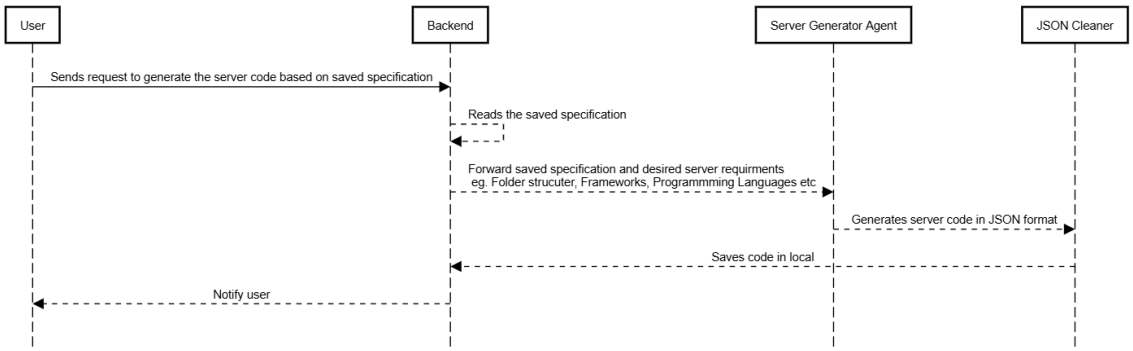

Figure 3.6 *Code Generation*

This input is processed by invoking the `specgenerator.generate_openAPI_spec` function. This function applies sophisticated logic to analyze the input and produce a detailed OpenAPI specification. The specification is then returned as a response, enabling users to review and validate the output. This interaction is fundamental to the workflow, serving as the starting point for subsequent stages such as server code generation and testing.

Server Code Generation (/generate Route)

The `/generate` route facilitates the automatic generation of server code from the OpenAPI specification. This process begins with the backend reading the specification from a predefined YAML file, ensuring that the input adheres to standard OpenAPI formatting. The `servergenerator.generate_server_code` function is then called to translate the specification into server code. This function generates code that aligns with the provided API structure, ensuring compliance with the specification's requirements. Once the server code is generated, it is converted into JSON format, making it suitable for storage and further interaction. The backend then stores this JSON representation, which can be used by users for testing, debugging, or deployment. This step bridges the gap between specification design and practical implementation.

Server Interaction (/tester Route)

The `/tester` route is a versatile interface designed for testing and interacting with the generated server code. Users can leverage this route to perform a variety of tasks, such as starting the server, retrieving logs, or addressing issues within the server code. The backend handles these interactions by utilizing a suite of predefined functions, such as `run_docker_compose`, `get_services_status`, and `send_api_request`. Each function is tailored to perform a specific operation, ensuring a streamlined and efficient workflow.

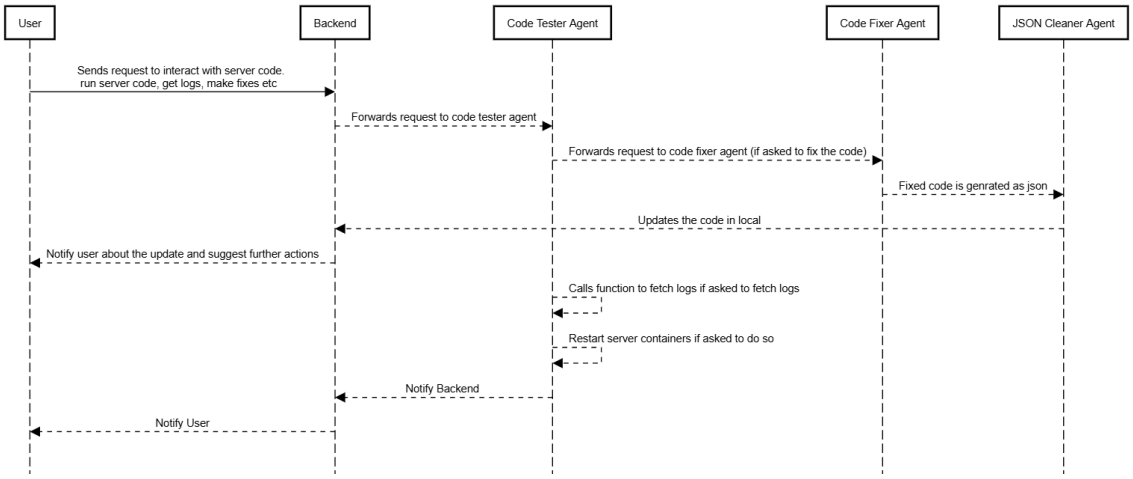

Figure 3.7 *Interaction With Server*

For example, the `run_docker_compose` function automates the process of starting Docker services associated with the server, while `get_services_status` provides real-time updates on the status of these services. Similarly, the `send_api_request` function enables users to test API endpoints by sending HTTP requests and evaluating the responses. The backend processes these commands, executes the required operations, and formats the output into user-friendly responses. This functionality is critical for validating the server code, identifying and fixing issues, and ensuring that the API operates as expected. Together, these interactions form a comprehensive testing and debugging environment, allowing users to seamlessly transition from code generation to practical deployment.

3.4 Evaluation Framework

To test the impact and capability of the system, a survey was conducted by getting direct feedback from industry practitioners. The tool's source code along with detailed usage instructions were provided to each participant. They were subsequently asked to complete a feedback form after using the tool to create a service.

The goal of this evaluation was to check and calculate how accurate the tools are in generating service code and how effective they are in debugging it. The system was first tested by each participant, after which a online survey was conducted to gather both quantitative and qualitative data.

3.4.1 Questionnaire Design

To effectively grade the system we carefully planned questions to capture both quantitative and qualitative insights from participants about their experience with the system. The interview had two main sections:

No.	Question
1	How many years of experience do you have in software development? Required to answer. Single choice.
2	How many years of experience do you have working with microservice architecture? Required to answer. Single choice.
3	Which programming languages are you most comfortable with?

Table 3.2 *Background Information Questions*

1. Participant's Background.

2. Questions for tool evaluation.

The survey begins with the questions about practitioner's overall experience with software development and with microservice architecture. The questions are in the table 3.2. These questions helped in making sense of the feedback based on their skill level.

The core of the interview was about the effectiveness of the system where participants were asked to rate how well the system translated the natural language prompts into OpenAPI specification, alignment of generated server code with specification, and the effectiveness of the system in saving time, helping with fixing the code and validating the server code using natural language prompts. It was a mix of 13 single-choice and 6 open-ended questions. The single choice questions include yes/no questions and statements with scale from 1 to 10, where 1 indicates strong disagreement and 10 indicates strong agreement. The open-ended questions focus on understanding developers' perception of the tool's advantages, limitations, and areas of improvement in their own words. This approach of using both open-ended and single-choice questions helped in ensuring that the feedback covers a wide range of aspects like code quality, folder structure, following of best practices, and production readiness.

3.4.2 Practitioners Selections

The participants were selected based on their experience in software development and familiarity with microservice architecture. A total of six developers participated in testing the system, representing a range of experience levels as shown in Table 3.5. These participants were selected using convenience sampling.

Most of the participants had more than 2 to 3 years of software development experience, with one of them having an experience of more than 10 years. Regarding the experience with microservices, most participants have limited experience i.e. 4 participants have 0-1 year of experience while 2 had 4 to 5 experience. This mixed selection of participants helped ensure that the result was well-rounded.

No.	Question
1	Did the tool successfully convert your natural language API description into an accurate OpenAPI specification? .
2	How well did the generated server code align with the OpenAPI specification and your initial description?
3	How much time did the tool save you compared to manually creating an API and server code?
4	How effective was the tool in helping you debug and fix issues in the generated code?
5	Did the tool's testing features (e.g., sending API requests) effectively validate the functionality of the generated API?
6	If No, what was the issue?
7	How effectively did the tool's code update and fix features work in resolving identified problems?
8	How does using this tool compare to your usual method of API and server code development in terms of speed?
9	How does the code quality produced by the tool compare to what you would typically write manually?
10	How would you rate the overall readability of the code generated by the tool?
11	Was the code organized in a logical manner (e.g., clear separation of concerns, modularity)?
12	Do you feel the tool-generated code adheres to best practices and coding standards (e.g., naming conventions, formatting)?
13	How likely are you to recommend the generated code to be used as part of a production system, purely based on its readability? .
14	How likely are you to replace or reduce your usage of other tools like Postman, Docker CLI, or your IDE if this tool offered comparable functionality in one interface?
15	What was the most significant advantage of using this tool? .
16	What was the most significant disadvantage or limitation of the tool?
17	Would you recommend this tool to other developers?
18	What improvements would you suggest for the tool?
19	How do you see this tool fitting into your regular development workflow?

Table 3.3 *Feedback Questions for the System*

Experience Type	Years of Experience	Number of Participants
Software Development	No Experience	1
	2–3 years	2
	4–5 years	1
	6–10 years	1
	10+ years	1

Table 3.4 *Participants' Years of Experience in Software Development*

Experience Type	Years of Experience	Number of Participants
Microservice Architecture	No Experience	4
	4-5 years	2

Table 3.5 *Participants' Years of Experience in Microservice Architecture*

By collecting this real-world data from industry practitioners having different experience levels, the evaluation provided a perfect measure of the tool's utility in real development environments which further helped in identifying both its strengths and areas of improvements.

3.4.3 Data Collection

The data collection process aimed to evaluate the system's usability and effectiveness based on participant feedback. The survey was conducted over a 2 month period, during which all six participants tested the system. Participants were emailed the usage instructions for the system, along with a link to the survey form. They were instructed to use the system first and then complete the survey form to ensure their responses reflected their hands-on experience.

To facilitate structured testing, participants were provided with a detailed step-by-step guide outlining how to set up and interact with the system. The tool usage sessions were self-paced, allowing participants to work at their convenience. On average, it took participants 51 minutes and 16 seconds to fill out the survey.

3.4.4 Data Analysis

The data collected from the survey was analyzed using both quantitative and qualitative methods. Quantitative responses, which mostly included ratings on a scale 1 to 10, were assessed by calculating the mean to identify central trends and overall satisfaction levels across categories such as API accuracy, code quality, and time-saving ability as shown in Figure 4.1. Modes were also calculated to identify the most frequent responses, that offered insights into the most common experiences participants had with the system. This helped to highlight recurring patterns in user feedback, providing a clearer picture of the system's overall performance. Qualitative data from open-ended questions was analyzed using thematic analysis to identify common patterns, suggestions, and feedback. These approaches allowed a detailed understanding of both the numerical data and written text shared by participants.

4 Preliminary Results

In this section, we present the results of data collected from the participants. The participants were assigned labels such as P1, P2, etc, and as shown in Figure 4.1 their year of experience is labeled with their respective provided ratings for each attribute of the system. We will discuss about data collected for each attribute of the system in the sections below. Due to the small sample size, these results should be interpreted as indicative rather than definitive.

4.1 RQ1: User Perceptions of Usability and Effectiveness

Users generally perceived the AI-driven system as more usable and effective compared to traditional manual coding methods. The mean rating for the tool's speed compared to usual methods was 7.8 out of 10, indicating a significant perceived improvement in efficiency. Moreover, system's ability in converting natural language API descriptions into OpenAPI specifications was positive, with all six participants reporting successful conversion. The mean rating for the alignment of generated server code with the OpenAPI specification was 7.0 out of 10 (mode = 8), indicating a high level of accuracy. Thematic analysis of qualitative responses revealed three primary themes:

Time Efficiency: Users consistently mentioned time-saving as a major advantage. One respondent noted, "It automates the whole process of building CRUD microservices, which saves a lot of time and effort."

Simplify Development: Users appreciated the tool's ability to handle multiple aspects of development. One developer(P2) commented, "It makes setting up CRUD microservices quick and easy by automating a lot of the boring, repetitive parts."

Workflow Integration: Users saw potential for the tool to streamline their development process. A developer(P4) commented, *"I would definitely see it as a great 'starter-pack' for initial API Design and specification phase, rapid prototyping and MVP development with API-First approach in Agile Development."*

However, some users, particularly those with more experience, expressed concerns about over-reliance on the tool. One senior developer(P3) cautioned, *"it is effective but can't rely totally".*

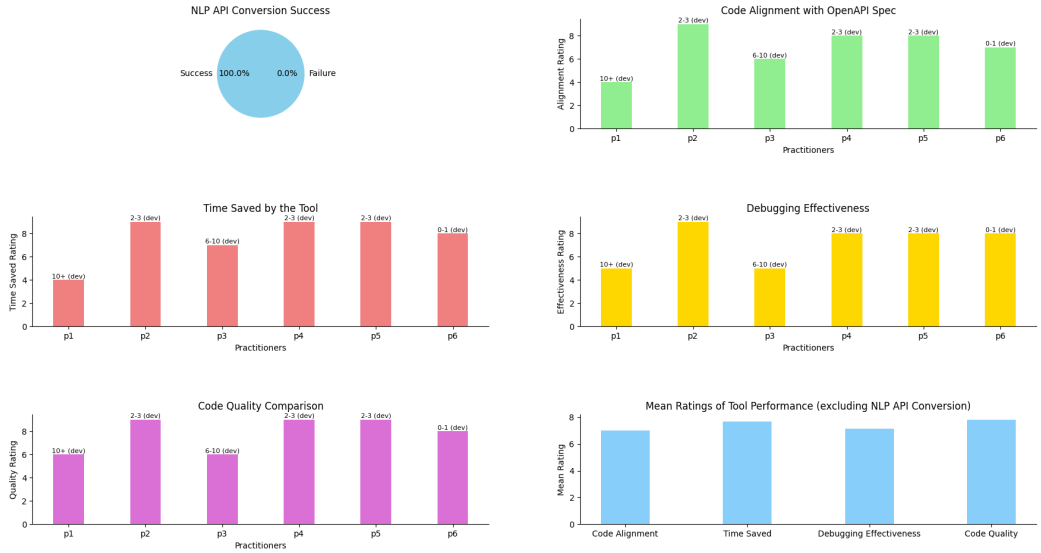

Figure 4.1 *Data Analysis*

4.2 RQ2: Reduction in Tool Switching

The system showed promise in reducing the need for switching between multiple development tools. The mean likelihood of replacing or reducing usage of other tools was 7.0 out of 10 with mode of 8, indicating a moderate to high potential for tool consolidation. Qualitative responses supported this finding.

Integrated Workflow: Users appreciated the tool's ability to combine multiple functions. A participant(P6) noted, "The ability to interact with Docker, execute server code, and send API requests using natural language prompts eliminates the need to switch between multiple tools."

Centralized Development: The tool's capacity to handle various aspects of development in one interface was frequently mentioned as an advantage.

Dependency Concerns: One user(P2) expressed concern about the tool's reliance on Docker, which could be a limitation in certain development environments.

However, more experienced developers expressed some reservations. One user(P3) commented that while the tool is effective, they "can't rely totally" on it, suggesting a need for integration with existing workflows rather than complete replacement.

4.3 RQ3: Impact on Manual Coding Efforts and Development Speed

The system demonstrated a significant positive impact on reducing manual coding efforts and increasing development speed. The mean rating for time saved compared to manual creation was 7.7 out of 10, indicating substantial time savings. Thematic analysis revealed two main themes:

Automation of Repetitive Tasks: Users appreciated the automation of CRUD operations and API documentation. A user(P2) stated, "It makes setting up CRUD microservices quick and easy by automating a lot of the boring, repetitive parts."

Rapid Prototyping: The system was seen as particularly valuable for quick prototyping and MVP development. A developer(P4) noted its usefulness in "rapid prototyping and MVP development with API-First approach."

However, some limitations were identified. A senior developer pointed out that for large-scale software development, the tool has "limited customization and flexibility," particularly for complex features like "custom authentication, authorization, data validation, error handling, and business logic."

5 Discussion

Several research efforts have been conducted to explore the use of LLMs for autonomous code generation, such as CodePori: A Large-Scale System for Autonomous Software Development Using Multi-Agent Technology [23]. CodePori relies on a multi-agent framework where different agents handle code generation and verification, but the verification process is done through simulations by other LLMs. This means that the system doesn't actually test the code in a real-world environment, which can lead to potential issues related to configuration mismatches, missing dependencies, or platform-specific bugs that aren't caught during testing. In contrast, this system [18] improves upon that approach by directly executing the generated code within the user's local environment. By running the code locally, this system ensures that the generated code functions as expected in the user's specific setup, addressing environment-specific issues right away. It also provides feedback based on logs generated and suggests fixes based on those logs, which helps the user resolve issues quickly. Unlike CodePori, which depends on abstract external verification done by LLM agent , this system offers a more reliable approach for validating code.

Moreover, participants involved in the interview, identified significant time savings, automation of repetitive tasks, and the ability to handle API design with code generation as the primary advantages. Participants with 2-3 years of experience highlighted the system's ability to automate the entire CRUD microservice generation process. Participants with more than 6 years of development found the system helpful in specification generation and documentation but suggested storing interaction history with the system. Another, suggestion from experienced participants was to improve the system's user interface.

6 Future Work

Based on the feedback there are several directions in which the CRUD Microservice Code Generator can be expanded and improved. This section describes areas of improvement to enhance the system's usability and functionality to better meet the needs of software developers and to ensure its applications in the actual development environment.

6.1 Enhancements to the System

To improve the system further the most immediate focus can be done on incorporating open-source models which can enhance flexibility and security. Integrating open-source alternatives to the current language model allows the tool to be adapted for a broader developer community and align more closely with open-source standards. Also, support for other languages and frameworks can be added like Python, Java, C#, and other popular programming languages. This will allow developers from diverse backgrounds to leverage the tool within their preferred technology stacks.

Another key area for future development is to improve system's ability to provide fixes because currently it does require further refinement in certain cases where user intervention is necessary to resolve more complex issues in the server code. In some situations, the fixes suggested by the agents are not sufficient to fully address the problem, and additional manual adjustments are needed. To improve this, more advanced models can be explored, or the current LLM models can be further trained to handle these edge cases. This could enhance the system's ability to recognize and suggest better fixes and improving its understanding of different types of issues in server code. By refining the model's capabilities to handle more complex scenarios autonomously, the system could reduce the need for user intervention and provide more effective solutions for code generation and debugging

Lastly, instead of only providing input through text, a voice-based interface could also be developed to enable users to perform the entire development process, from generating code to deploying it, using spoken commands. This would further simplify the development workflow, making the tool more accessible to users.

6.2 Expansion to Other Use Cases

The system's ability to fix and modify the codebase while maintaining the file structure can be expanded to larger projects by dividing the codebase into smaller chunks such that a single LLM agent can manage a chunk. Then, these agents can work

in teams to implement fixes or do upgrades. Additionally, more functions can be added to the LLMs tool-calling feature like complex Docker commands or other system commands commonly used in development environments, such as Git commands or cloud deployment commands. This would allow users to interact with the system using natural language prompts and users don't have to remember complex commands to complete the development process.

These advancements aim to increase the tool's adaptability and reduce the manual effort required in API and server code development, making it a more powerful and intelligent developer assistant.

References

[1] Stenly Ibrahim Adam, Jimmy H Moedjahedy, and Jeremiah Maramis. "RESTful Web Service Implementation on Unklab Information System Using JSON Web Token (JWT)". In: *2020 2nd International Conference on Cybernetics and Intelligent System (ICORIS)*. 2020, pp. 1–6. DOI: 10.1109/ICORIS50180.2020.9320801.

[2] Hussam Alkaissi and Samy I McFarlane. "Artificial hallucinations in ChatGPT: implications in scientific writing". In: *Cureus* 15.2 (2023).

[3] Kapil Bakshi. "Microservices-based software architecture and approaches". In: *2017 IEEE Aerospace Conference*. 2017, pp. 1–8. DOI: 10.1109/AERO.2017.7943959.

[4] Sandra Casas et al. "Uses and applications of the OpenAPI/Swagger specification: a systematic mapping of the literature". In: *2021 40th International Conference of the Chilean Computer Science Society (SCCC)*. 2021, pp. 1–8. DOI: 10.1109/SCCC54552.2021.9650408.

[5] Rui Chen, Shanshan Li, and Zheng Li. "From Monolith to Microservices: A Dataflow-Driven Approach". In: *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 2017, pp. 466–475. DOI: 10.1109/APSEC.2017.53.

[6] Xi Chen, Yuebin Liao, and Wei Yu. "Generative AI in Higher Art Education". In: *2024 6th International Conference on Computer Science and Technologies in Education (CSTE)*. 2024, pp. 135–140. DOI: 10.1109/CSTE62025.2024.00032.

[7] Brajesh. De. *API Management : An Architect's Guide to Developing and Managing APIs for Your Organization*. eng. 2nd ed. 2023. Berkeley, CA: Apress, 2023. ISBN: 979-88-6880-054-2.

[8] Sherry Ding and Veda Raman. "Harness the Power of Generative AI in Healthcare with Amazon AI/ML Services". In: *2024 IEEE 12th International Conference on Healthcare Informatics (ICHI)*. 2024, pp. 490–492. DOI: 10.1109/ICHI61247.2024.00070.

[9] Mario Dudjak and Goran Martinović. "An API-first methodology for designing a microservice-based Backend as a Service platform". eng. In: *Information technology and control* 49.2 (2020), pp. 206–223. ISSN: 1392-124X.

[10] Javad Ghofrani and Daniel Lübke. "Challenges of Microservices Architecture: A Survey on the State of the Practice." In: *ZEUS* 2018 (2018), pp. 1–8.

[11] Juyong Jiang et al. "A Survey on Large Language Models for Code Generation". In: *arXiv preprint arXiv:2406.00515* (2024).

[12] Carolina Centeio Jorge, Myrthe L. Tielman, and Catholijn M. Jonker. "Artificial Trust as a Tool in Human-AI Teams". In: *2022 17th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. 2022, pp. 1155–1157. DOI: 10.1109/HRI53351.2022.9889652.

[13] Kuldeep Singh Kaswan et al. "Generative AI: A Review on Models and Applications". In: *2023 International Conference on Communication, Security and Artificial Intelligence (ICCSAI)*. 2023, pp. 699–704. DOI: 10.1109/ICCSAI59793.2023.10421601.

[14] Koren Lazar et al. *SpeCrawler: Generating OpenAPI Specifications from API Documentation Using Large Language Models*. 2024. arXiv: 2402.11625 [cs.CL]. URL: https://arxiv.org/abs/2402.11625.

[15] Alexander Lercher. "Managing API Evolution in Microservice Architecture". In: *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2024, pp. 195–197. DOI: 10.1145/3639478.3639800.

[16] James Lewis and Martin Fowler. "Microservices: a definition of this new architectural term". In: *MartinFowler. com* 25.14-26 (2014), p. 12.

[17] Jiawei Liu et al. *Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation*. 2023. arXiv: 2305.01210 [cs.SE]. URL: https://arxiv.org/abs/2305.01210.

[18] Author Name. *Code for service generation and instructions*. 2024. URL: https://github.com/sirbh/code-gen.

[19] Lauren Olson. "Custom Developer GPT for Ethical AI Solutions". In: *2024 IEEE/ACM 3rd International Conference on AI Engineering – Software Engineering for AI (CAIN)*. 2024, pp. 282–283.

[20] *OpenAPI Specification*. https://github.com/OAI/OpenAPI-Specification.

[21] Christian Posta. *Microservices for Java Developers: A Hands-on Introduction to Frameworks and Containers*. O'Reilly Media, 2016.

[22] RV Rajesh. *Spring Microservices*. Packt Publishing Ltd, 2016.

[23] Zeeshan Rasheed et al. *CodePori: Large-Scale System for Autonomous Software Development Using Multi-Agent Technology*. 2024. arXiv: 2402.01411 [cs.SE]. URL: https://arxiv.org/abs/2402.01411.

[24] José Matías Rivero et al. "MockAPI: An Agile Approach Supporting API-first Web Application Development". In: *Web Engineering*. Ed. by Florian Daniel, Peter Dolog, and Qing Li. Springer Berlin Heidelberg, 2013.

[25] Yamina Romani, Okba Tibermacine, and Chouki Tibermacine. “Towards Migrating Legacy Software Systems to Microservice-based Architectures: a Data-Centric Process for Microservice Identification”. In: *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. 2022, pp. 15–19. DOI: 10.1109/ICSA-C54293.2022.00010.

[26] Malik Sallam. “ChatGPT utility in healthcare education, research, and practice: systematic review on the promising perspectives and valid concerns”. In: *Healthcare*. Vol. 11. 6. MDPI. 2023, p. 887.

[27] Vivian Nguyen Saša Baškarada and Andy Koronios. “Architecting Microservices: Practical Opportunities and Challenges”. In: *Journal of Computer Information Systems* 60.5 (2020), pp. 428–436. DOI: 10.1080/08874417.2018.1520056. eprint: https://doi.org/10.1080/08874417.2018.1520056. URL: https://doi.org/10.1080/08874417.2018.1520056.

[28] Saurabh. *LLM-Generated Microservice Implementations from RESTful API Definitions*. DOI: 10.5281/zenodo.14505669. 2024. DOI: 10.5281/zenodo.14505669. URL: https://doi.org/10.5281/zenodo.14505669.

[29] Divarshana Saxena and Biswajit Bhowmik. “Paradigm Shift From Monolithic to Microservices”. In: *2023 IEEE International Conference on Recent Advances in Systems Science and Engineering (RASSE)*. 2023, pp. 1–7. DOI: 10.1109/RASSE60029.2023.10363466.

[30] *Swagger: The Open Source API Framework*. https://swagger.io/.

[31] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. “Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation”. In: *IEEE Cloud Computing* 4.5 (2017), pp. 22–32.

[32] Jiexin Wang et al. *Enhancing Large Language Models for Secure Code Generation: A Dataset-driven Study on Vulnerability Mitigation*. 2023. arXiv: 2310.16263 [cs.SE]. URL: https://arxiv.org/abs/2310.16263.

[33] Junfeng Wang, Qiliang Yang, and Yin Chen. “A Large Language Model–based Approach for automatically Optimizing BIM”. In: *2024 43rd Chinese Control Conference (CCC)*. 2024, pp. 8518–8523. DOI: 10.23919/CCC63176.2024.10661504.

[34] Wayne Xin Zhao et al. *A Survey of Large Language Models*. 2024. arXiv: 2303.18223 [cs.CL]. URL: https://arxiv.org/abs/2303.18223.

[35] Daniel Zimmermann and Anne Koziolek. “GUI-Based Software Testing: An Automated Approach Using GPT-4 and Selenium WebDriver”. In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. 2023, pp. 171–174. DOI: 10.1109/ASEW60602.2023.00028.

APPENDIX A. Example OpenAPI Specification Created Using System

1 OpenAPI Specification for Product Service API

The following OpenAPI 3.0 specification generated using the system defines the Product Service API, which provides endpoints for managing products in an inventory system. It includes operations to list, create, retrieve, update, and delete products. The specification is structured in YAML format, detailing each endpoint, request and response bodies, and associated data models used within the API.

```yaml
openapi: 3.0.0
info:
  title: Product Service API
  description: API for managing products
  version: 1.0.0
servers:
  - url: https://api.example.com/v1
paths:
  /products:
    get:
      summary: List all products
      responses:
        '200':
          description: A list of products
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Product'
    post:
      summary: Create a new product
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ProductInput'
```

```
29      responses:
30        '201':
31          description: Product created successfully
32          content:
33            application/json:
34              schema:
35                $ref: '#/components/schemas/Product'
36  /products/{productId}:
37    get:
38      summary: Get a product by ID
39      parameters:
40        - name: productId
41          in: path
42          required: true
43          schema:
44            type: string
45      responses:
46        '200':
47          description: A single product
48          content:
49            application/json:
50              schema:
51                $ref: '#/components/schemas/Product'
52        '404':
53          description: Product not found
54    put:
55      summary: Update a product by ID
56      parameters:
57        - name: productId
58          in: path
59          required: true
60          schema:
61            type: string
62      requestBody:
63        required: true
64        content:
65          application/json:
66            schema:
67              $ref: '#/components/schemas/ProductInput'
68      responses:
69        '200':
70          description: Product updated successfully
```

```
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Product'
        '404':
          description: Product not found
    delete:
      summary: Delete a product by ID
      parameters:
        - name: productId
          in: path
          required: true
          schema:
            type: string
      responses:
        '204':
          description: Product deleted successfully
        '404':
          description: Product not found
components:
  schemas:
    Product:
      type: object
      properties:
        id:
          type: string
        name:
          type: string
        description:
          type: string
        price:
          type: number
          format: float
    ProductInput:
      type: object
      properties:
        name:
          type: string
        description:
          type: string
        price:
          type: number
```

```
113           format: float
114       required:
115         - name
116         - price
```

APPENDIX B. Example Microservice Code Created Using System

This section provides an example of microservice code generated by the system for a product management service. The code demonstrates how various components of a microservice, such as routing, controllers, and database interaction, can be automatically created using predefined OpenAPI specification.

The example focuses on a simple product service API, allowing users to perform CRUD (Create, Read, Update, Delete) operations on products. It showcases the integration of Prisma ORM for database management with a PostgreSQL database, Express.js for server-side routing

1 Server Code

Code generated for server.js file

```
const express = require('express');
const app = express();
const routes = require('./routes');
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();

app.use(express.json());
app.use('/v1', routes);

const PORT = process.env.PORT || 3001;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

2 Package Management File

Here is an example of a generated JSON configuration file:

```
{
  "name": "product-service-api",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
```

```
  },
  "dependencies": {
    "@prisma/client": "^3.0.0",
    "express": "^4.17.1",
    "product-service-api": "file:"
  },
  "devDependencies": {
    "prisma": "^3.0.0"
  }
}
```

3 Dockerfile Example

Here is an example of a Dockerfile for the same project:

```
FROM node:14

WORKDIR /app

COPY package.json .
# COPY package-lock.json .

RUN npm install

COPY . .

RUN npx prisma generate

CMD ["node", "server.js"]
```

4 Docker Compose Configuration

Here is an example of a generated ‘docker-compose.yml‘ file for setting up a PostgreSQL database and an application server:

```
version: '3.8'

services:
  database:
    image: postgres:13
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
```

```
      POSTGRES_DB: mydb
    ports:
      - '5432:5432'
    volumes:
      - db_data:/var/lib/postgresql/data

  server:
    build: .
    ports:
      - '3001:3001'
    depends_on:
      - database
    environment:
      - DATABASE_URL=postgresql://user:password@database:5432/mydb?schema=public
      - PORT=3001

volumes:
  db_data:
```

5 Environment Variables

Here is an example of environment variables used for configuring the server:

```
DATABASE_URL=postgresql://user:password@database:5432/mydb?schema=public
PORT=3001
```

6 Express.js Routes

Here is an example of route definitions for handling product-related API requests:

```
const express = require('express');
const router = express.Router();
const productController = require('../controllers');

// Define routes
router.get('/products', productController.getAllProducts);
router.post('/products', productController.createProduct);
router.get('/products/:productId', productController.getProductById);
router.put('/products/:productId', productController.updateProductById);
router.delete('/products/:productId', productController.deleteProductById);

module.exports = router;
```

7 Prisma Schema

Here is an example of a Prisma schema file for defining a ‘Product‘ model:

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model Product {
  id          String  @id @default(cuid())
  name        String
  description String?
  price       Float
}
```

8 Controller File

Here is an example of a controller for managing products in your application:

```
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();

// Controller functions
exports.getAllProducts = async (req, res) => {
  try {
    const products = await prisma.product.findMany();
    res.json(products);
  } catch (error) {
    res.status(500).json({ error: 'Internal Server Error' });
  }
};

exports.createProduct = async (req, res) => {
  try {
    const { name, description, price } = req.body;
    const product = await prisma.product.create({
      data: { name, description, price }
    });
```

```
20      res.status(201).json(product);
21    } catch (error) {
22      res.status(500).json({ error: 'Internal Server Error' });
23    }
24  };
25
26  exports.getProductById = async (req, res) => {
27    try {
28      const { productId } = req.params;
29      const product = await prisma.product.findUnique({
30        where: { id: productId }
31      });
32      if (product) {
33        res.json(product);
34      } else {
35        res.status(404).json({ error: 'Product not found' });
36      }
37    } catch (error) {
38      res.status(500).json({ error: 'Internal Server Error' });
39    }
40  };
41
42  exports.updateProductById = async (req, res) => {
43    try {
44      const { productId } = req.params;
45      const { name, description, price } = req.body;
46      const product = await prisma.product.update({
47        where: { id: productId },
48        data: { name, description, price }
49      });
50      res.json(product);
51    } catch (error) {
52      res.status(404).json({ error: 'Product not found' });
53    }
54  };
55
56  exports.deleteProductById = async (req, res) => {
57    try {
58      const { productId } = req.params;
59      await prisma.product.delete({
60        where: { id: productId }
61      });
```

```
62      res.status(204).end();
63    } catch (error) {
64      res.status(404).json({ error: 'Product not found' });
65    }
66  };
```