Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Fakultät für Ingenieurwesen
Facoltà di Ingegneria
Faculty of Engineering

Master in Software Engineering

Master Thesis

Automating Software Performance Tests using Agentic AI

Candidate: Elias Binder

Supervisor: Andrea Janes

July, 2025

Abstract

Motivation Software performance testing is essential for ensuring application reliability and user satisfaction, especially in complex, fast-evolving systems. Traditional performance testing methods are time-consuming, require expert knowledge, and often fail to scale with modern development practices.

Problem statement Existing performance testing approaches lack automation and adaptability, making them inefficient and inaccessible for many development teams.

Approach This thesis proposes a fully automated performance testing framework using agentic AI and large language models. A desktop application was developed that integrates LangGraph agents, MCP servers, and the PPTAM framework to autonomously generate, execute, and analyze performance tests based on existing project artifacts.

Results The tool was successfully evaluated on two real-world microservice-based applications, demonstrating its ability to produce meaningful test scenarios, execute them across environments, and generate insightful performance reports with minimal human input.

Conclusions This work shows that agentic AI can significantly reduce the manual effort required for performance testing, making it more accessible and scalable. Developers facing similar challenges can adopt this approach to streamline their testing workflows and improve software quality without deep performance engineering expertise.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Artificial Intelligence (AI) has swiftly transitioned from a specialised research domain to a transformative force reshaping industries worldwide [1]. Among the most significant advancements is the development of novel AI architectures, such as the transformer architecture, which has revolutionised natural language processing (NLP) and influenced a broad spectrum of machine learning applications [2]. These innovations are increasingly being integrated into software engineering workflows, supporting tasks such as code generation, documentation, and automated testing [3]. As these technologies continue to mature, they unlock new possibilities for automating complex processes that have traditionally relied on substantial human expertise [4].

One such process is software performance testing, a vital phase in the software development lifecycle that ensures applications meet performance benchmarks under varying conditions. While automation in performance testing is not a new concept, the incorporation of LLMs introduces groundbreaking capabilities for generating, executing, and analysing performance tests with minimal human oversight.

1.1 Motivation

Software performance testing is a critical yet challenging aspect of modern quality assurance, especially as applications become more complex and user expectations rise [5]. Traditional approaches to performance testing are often time-consuming and require specialised domain knowledge, leading to bottlenecks and inadequate coverage [6]. Moreover, the quickly evolving nature of software systems means that performance characteristics can shift with each release, making static test suites increasingly irrelevant [6].

The rise of microservices and cloud-native applications has further complicated performance testing due to their nested interdependencies [5] [6]. This necessitates a more dynamic testing approach that can adapt to changes in the system. Recent advancements in natural language processing, particularly in agentic AI and agent-to-agent workflows, present a unique opportunity to revolutionise performance testing. By employing intelligent automation, we can enhance testing efficiency, reduce the need for extensive human expertise, increase coverage, and ultimately produce more reliable software systems [7]. Addressing these challenges is vital for meeting the demands of modern software development and ensuring optimal application performance.

1.2 Objective

The primary objective of this thesis is to investigate and demonstrate the feasibility of leveraging agentic AI and Large Language Models to fully automate the end-to-end software performance testing process. This research aims to bridge the gap between the theoretical potential of AI-driven testing and practical implementation by developing and evaluating a comprehensive tool that can mostly autonomously generate, execute, and analyse performance tests with minimal human intervention.

Specifically, this work seeks to achieve the following key goals:

- **Automated Test Scenario Generation:** Develop an AI system to automatically generate meaningful performance test scenarios, given already available sources, such as the codebase of the system under test, access its Jira/GitHub issues, etc. This includes understanding application architecture, identifying critical performance paths, and creating realistic load patterns that reflect actual usage scenarios. The system should be able to generate diverse test cases that cover various performance testing strategies, such as load testing, stress testing, and scalability testing.

- **Autonomous Test Execution:** Implement an agentic AI framework that can independently execute the generated performance tests without human intervention. This involves orchestrating test environments, managing test data, monitoring system behaviour during test execution, and handling unexpected scenarios or failures that may arise during testing.

- **Intelligent Result Analysis:** Create an AI-powered analysis component capable of interpreting performance test results, identifying bottlenecks, detecting anomalies, and providing actionable insights for performance optimisation. The system should be able to correlate performance metrics with the strategy of the performance test and suggest potential improvements or areas of concern.

- **Validation of Effectiveness:** Evaluate the developed tool's effectiveness by comparing its performance against traditional testing approaches. This includes assessing the quality of generated test scenarios, the accuracy of performance bottleneck identification, and the practical utility of AI-generated insights for software optimisation.

- **Exploration of Limitations and Opportunities:** Identify the current limitations of LLM and agentic AI approaches in performance testing contexts, while also exploring opportunities for future enhancement and broader application within the software testing domain.

Through achieving these objectives, this thesis aims to contribute to the growing body of knowledge on AI-assisted software engineering while providing practical insights into the application of cutting-edge AI technologies to solve real-world software testing challenges.

1.3 Approach

This research adopts a proof-of-concept approach through the development of a comprehensive desktop application that demonstrates the practical application of agentic AI and agent-to-agent workflows to automated performance testing. The application serves as an integrated environment where developers can leverage AI-driven automation for the complete performance testing workflow while maintaining minimal human intervention requirements. A key architectural decision is the utilisation of readily available development resources to maximise the contextual understanding of the AI system while minimising additional overhead for developers. The application is designed to seamlessly connect to and analyse multiple information sources that are typically present in modern software development environments, including the software codebase itself, project management systems such as GitHub issues and Jira tickets, API documentation in the form of Swagger or OpenAPI definition files, and other relevant project artefacts. This integration strategy ensures that the underlying LLM has access to exceptional detail about the software system's architecture, functionality, known issues, and intended behaviour, enabling the system to understand implementation details, identify critical execution paths, and recognise patterns that may impact performance. The overall goal is to reduce the human intervention required throughout the performance testing lifecycle by intelligently leveraging the wealth of information available in existing development artefacts. The proof-of-concept implementation follows an iterative development approach, with emphasis on creating a user-friendly interface that abstracts the complexity of the underlying AI systems while providing transparency into the automated decision-making processes, ensuring that sophisticated performance analysis becomes accessible to development teams without dedicated performance testing specialists.

1.4 Contributions of this thesis

This thesis makes the following contributions:

- The definition of the problem and requirements, which are necessary to address
- The architecture and design of the developed solution
- The developed prototype *Performance Testing Agent* available at `https://github.com/EliasBinder/Performance-Testing-Agent`
- The developed *Langgraph Agents* in Python available at `https://github.com/EliasBinder/Performance-Testing-Agent-Langgraph`
- An MCP Server to explore the code base *Codebase Explorer MCP Server* available at `https://github.com/EliasBinder/MCPServer-Codebase-Explorer` *Swagger Explorer MCP Server* available at `https://github.com/EliasBinder/MCPServer-Swagger-Explorer` *PPTAM* wrapped with an HTTP Server available at `https://github.com/EliasBinder/PPTAM-With-HTTP-Wrapper`
- The evaluation of the developed prototype
- The discussion of the advantages and limitations of the developed prototype

1.5 Structure of the thesis

This thesis is organised into seven sections. It begins with an introduction (Sect. 1), which outlines the problem, states the objectives, and provides an overview of the approach used.

The next section presents the problem statement (Sect. 2), detailing the requirements envisioned for the proposed tool. Following that, an overview of the state of the art (Sect. 3) is provided in the form of a mapping study, focusing on specific types of tools and the technologies involved.

The following section (Sect. 4), outlines the proposed solution and details the system's architecture from different perspectives. The next section 5, describes the methods used to validate and test the developed solution. After that, section 6 presents the evaluation results and discusses their implications in relation to the initial research question. Finally, section 7 summarises the key activities undertaken, emphasises their importance, addresses the technical challenges faced, and suggests directions for future work.

Chapter 2

Problem Statement

This chapter examines the limitations of traditional performance testing, highlighting inefficiencies and scalability issues in manual methods. To address these challenges, a novel tool is introduced that uses agentic artificial intelligence to automate the creation, execution, and analysis of performance test scenarios.

2.1 Problems with traditional approaches to performance testing

In the realm of software engineering, ensuring the performance and scalability of applications under varying load conditions is a critical concern. Performance testing, a key technique in this domain, traditionally involves manually designing test scenarios, executing them, and interpreting the results, which often reveals itself to be a time-consuming process that is error-prone and difficult to scale [8]. As systems grow more complex and the demand for rapid deployment increases, the limitations of manual performance testing become increasingly apparent [8]. This thesis addresses these challenges by introducing a novel tool that leverages agentic artificial intelligence to automate the creation, execution, and analysis of performance test scenarios, thereby enhancing efficiency, accuracy, and adaptability in performance testing workflows.

Performance testing encompasses various methods, including load testing, stress testing, and scalability testing, all designed to evaluate responsiveness, stability, and resource utilisation [9]. Selecting the correct type of performance test requires specialised knowledge and experience in the field. A performance test must always fulfil one or several goals which can be formulated as questions, e.g. *What is the maximum load my system can handle?* or *Can my system withstand the average load throughout the whole day?*. The selection of goals also requires a deep understanding of the software system under test's architecture. Regarding a microservice architecture, for example, scalability tests are typically a much higher concern, as these types of architectures require a more complex infrastructure setup, which, if misconfigured, can quickly lead to bottlenecks and other performance issues.

Traditionally, performance testing involves manually designing test scenarios that simulate user behaviour, configuring test environments, executing tests using tools like Apache JMeter or LoadRunner, and then analysing the results by hand to identify performance concerns by focusing on preselected metrics [8]. Regarding industry application, more and more companies are integrating performance tests into their CI/CD pipelines to be automatically executed and evaluated when the codebase changes, which provides evidence that at least some degree of automation is necessary in the area.

2.2 Requirements for the suggested solution

To accommodate this, the starting point of this thesis is formulated in the form of requirements, following the schema presented in [10]. Specifically, the criteria will be divided into goal, domain, process/product, and design requirements. Moreover, requirements are formulated using the schema proposed by [11].

In the context of requirements engineering, the classification into goal, domain, process, and design requirements helps structure and clarify the intent, constraints, and implementation of a system. Each category serves a distinct purpose:

- **Goal Requirements** describe the high-level objectives or desired outcomes that the system should achieve. These are often abstract and stakeholder-driven, focusing on why the system is being developed. For example, "Improve user engagement on the platform" is a goal requirement.
- **Domain Requirements** capture the rules, constraints, and facts about the environment in which the system operates. These are typically derived from the real-world context and are often non-negotiable. For instance, "The system must comply with GDPR regulations" is a domain requirement.
- **Process/Product Requirements** specify how the development or operational processes should be carried out. These might include methodologies, workflows, or standards to be followed during the system's lifecycle. An example would be, "The development must follow Agile Scrum methodology."
- **Design Requirements** are more concrete and technical, detailing how the system should be built to meet the goals and constraints. These include architectural decisions, interface specifications, and performance criteria. For example, "The system must support at least 10,000 concurrent users" is a design requirement.

This layered approach ensures that the system is not only technically sound but also aligned with the needs of stakeholders, the industry, and environmental constraints.

Regarding the development of the tool discussed in this thesis to fully automate performance testing, the **goal** requirement is:

1. As a *developer*, I do not want to *deal with performance issues* so that *I can focus on providing value for my clients.*

The **domain** requirements are:

1. As a *developer*, I want to *avoid spending time with performance tests* so that *I can prioritise adding new features over fixing performance-related bugs.*
2. As a *developer*, I want to *get clear instructions on how to fix performance issues* so that *I can be productive and quickly solve them.*
3. As a *customer* I want that *performance issues are resolved quickly* so that *I can benefit more from the software.*
4. As a *customer*, I want *performance issues to be identified and fixed before deployment* so that *they do not affect my work.*

5. As a *developer*, I want to *re-run performance tests without redefining them* so that *I can validate changes quickly*.

The **product** requirements are:

1. As a *developer*, I want *an AI based solution* so that *I do not have to manually create, execute, and analyse performance tests.*
2. As a *developer*, I want to *be able to configure the solution* so that *I can adapt it to different types of software I am developing.*
3. As a *developer*, I want to *understand how the AI solution works* so that *I can intervene if it does something I do not need.*
4. As a *developer*, I want to *obtain a report of the results of the solution* so that *I can integrate the suggestions in the code I am developing.*
5. As a *developer*, I want to *re-execute previously generated performance tests* so that *I can validate performance after code changes.*

The **design** requirements are (formulated using the terms recommended in [12]):

1. The solution must be implemented in Python.
2. The solution must require as little user input as possible and rely mainly on already available information, such as the codebase or JIRA issues
3. The test scenarios must be created using agentic AI, implemented using LangGraph[1].
4. The agent must use PPTAM[2] [13] to execute the performance tests in local, staging and production environments.
5. The result analysis must generate a final report indicating weaknesses and bottlenecks in the system.
6. The solution must provide human-in-the-loop checkpoints to allow user validation of intermediate results.
7. The solution must be independent from the architecture of the software under test.

To further define the scope of the proof-of-concept solution and to avoid insane complexity by covering all sorts of software, the proposed tool must specifically test web servers, focusing on the automated testing of HTTP endpoints. While many existing solutions, such as JMeter or LoadRunner, require significant expertise to operate effectively [14], this tool must embed deep domain knowledge of performance testing. It must be designed to guide users through the entire process, from scenario creation to execution and result analysis, without requiring expert intervention. This enables true end-to-end automation, eliminating the need for performance-testing experts to intervene in the process. Therefore, it autonomously gathers insights about the system under test and proceeds with minimal

[1] `https://www.langchain.com/langgraph`
[2] `https://github.com/pptam/pptam-tool`

manual input. Nevertheless, it must incorporate human-in-the-loop mechanisms, allowing users to review and adjust the process at key stages and gather minimal information from the user, e.g. to select a goal, since the tool cannot know why the user wants to execute a performance test in the first place. To ensure accessibility and ease of use, the tool must be delivered as a desktop application with a user-friendly graphical interface. Furthermore, it is built to be architecture-independent, functioning seamlessly regardless of whether the system under test follows a monolithic, microservices, or any other architectural style.

Chapter 3

Review of the state of the art

This chapter summarises the current research in AI-driven software performance testing. It positions the proposed solution within the broader academic and industry context by highlighting relevant methods and technologies for performance test automation. Through a systematic mapping study, the chapter examines how artificial intelligence, particularly agentic AI and large language models, is used to automate various stages of the performance testing process. By analysing trends and gaps in the literature, it sets the groundwork for understanding the significance of the approach developed in this thesis.

3.1 Preparation of a Systematic Mapping Study

To ground the development of the proposed agentic AI-based tool for automating performance testing, it is essential to first understand the broader landscape of artificial intelligence (AI) applications in software testing. This chapter, therefore, presents a systematic mapping study that explores the current state of research on AI-driven automation in performance testing. While the focus of this thesis is on agentic AI, the scope of this study is intentionally broader, encompassing a wide range of AI techniques, including machine learning, deep learning, and heuristic-based approaches.

By surveying the wider field, this chapter aims to identify existing trends, gaps, and opportunities in the automation of performance testing. This broader perspective not only contextualises the novelty of our agentic approach but also highlights how it fits within and advances the current body of knowledge.

It also serves as proof to justify the exploration of other AI. The insights gained from this mapping study not only illuminate the current landscape of AI-driven automation in performance testing but also underscore the need for more autonomous, adaptive solutions, thereby justifying the relevance and timeliness of the agentic AI-based approach discussed in this thesis.

3.2 Related secondary literature

The integration of artificial intelligence (AI) into software testing has gained substantial research interest, with several systematic reviews examining this intersection. Trudova et al. [15] conducted a systematic literature review that catalogued AI techniques applied across various software testing activities, emphasizing their role in test automation. Similarly, Battina [16] provided a comprehensive review of AI applications in test case generation, defect prediction, test case prioritization, and Android testing. However, their taxonomy notably underrepresented performance testing automation.

In the context of machine learning-assisted performance testing, Moghadam et al. [17] found that supervised learning—particularly neural networks—is commonly used to generate inputs for system, GUI, and performance testing scenarios.

Despite these efforts, a significant gap remains in the systematic exploration of AI-driven performance testing automation. Existing reviews predominantly focus on functional testing and related tasks, offering limited insight into how AI can support the creation, execution, and analysis of performance test scenarios. This study addresses that gap by systematically mapping AI techniques across the entire performance testing lifecycle, offering a comprehensive overview of current research and highlighting directions for future investigation.

3.3 Research method

This research follows the guidelines for conducting a Systematic Mapping Study (SMS) proposed by Petersen et al. [18] to explore the current landscape of research on the use of artificial intelligence in automating software performance testing.

3.3.1 Research Questions

This study is guided by the following research questions, grouped into thematic areas. A funnel approach is employed to establish research questions, beginning with an overall understanding of how the research field has developed over the past years and what are current points of particular interest, then focusing on the automation of performance tests in detail, and extracting state-of-the-art metrics and quantification methods to gain insight into whether the adoption of AI in software performance testing is successful.

- **RQ1: How did the research field evolve over the past years?**
  - *Rationale:* This question explores how the research field has evolved since AI became mainstream and how the release of advanced AI computing methods, such as machine learning or large language models, influenced its growth.
- **RQ2: What are the existing approaches to automating performance testing in software systems using AI?**
  - *Rationale:* This question aims to explore how AI adoptions in software testing are making their way into the field of performance testing. The goal here is to catalogue both traditional and AI-based automation techniques, providing a baseline for comparison.
- **RQ3: How effective are AI-based approaches compared to other performance testing baselines?**
  - *Rationale:* This question evaluates the value of AI in performance testing. It seeks evidence of improvements in efficiency, accuracy, scalability, or cost effectiveness. This is crucial for justifying the use of AI models in practice.

- **RQ4: What metrics and methods are used to evaluate the success of AI-driven performance testing?**
  - *Rationale:* Understanding how success is measured helps assess the comparability of existing studies. It also informs future research by highlighting which metrics are most meaningful (e.g., response time reduction, test coverage, false positive rate).

3.3.2 Study Selection

This section focuses on defining strict boundaries of what falls within the scope of this research and what is being excluded. Therefore, inclusion and exclusion criteria were determined to act as a decision framework for the literature included in the analysis for this study.

Inclusion criteria: *Studies that discuss the application of Artificial Intelligence (AI), including Large Language Models (LLMs), in aspects of software performance testing and their automation.*

Given the inclusion criteria, the following exclusion criteria were derived:

- Study does not use AI for any step of performance testing, i.e.
  - Generation of performance test scenarios
  - Execution of the performance test
  - Interpretation of the performance test results
- Study is not peer-reviewed (e.g., blog posts, white papers, opinion pieces).
- Study is not available online
- Study has already been published elsewhere with a different name
- Study is a secondary study
- Study is not in English
- Study is published before 2018
- Studies that do not provide sufficient methodological detail or empirical evidence to support their claims

3.3.3 Search Strategy

In order to find studies that match the criteria defined in the section "Study Selection" and the research questions, a search query must first be constructed. Therefore, the most prevalent keywords must be found for every aspect of this study. These keywords are listed below:

- **Performance Testing**
  - *Rationale:* Find research that is about performance testing. It also makes sense to check for keywords like "load testing", "stress testing", "scalability testing", "throughput testing", or "latency testing" to also retrieve studies that do not focus on performance testing in general but instead on only one of its many aspects.

- **Artificial Intelligence**
  - *Rationale:* Filter out all publications that do not use artificial intelligence in any form. This includes terms like "artificial intelligence", "AI", "machine learning", "deep learning", "agentic AI", "LLM", "large language model", "language model" or "GPT".
- **Restriction to Software**
  - *Rationale:* Performance testing is also applied in non-software domains such as civil engineering (e.g., bridges, buildings). To ensure relevance, the search query must restrict results to the software engineering domain. This was achieved by including the terms "software system", "software application", "software engineering", or "software testing" and excluding unrelated domains. Additionally, including "microservices" and "web services" provided research with relevance, as performance tests often target these types of applications.
- **Publication Year**
  - *Rationale:* The search will be limited to studies published in 2018 or later, as this marks the release of the first version of BERT and the beginning of the modern AI surge, providing a foundation for subsequent models like RoBERTa, DistilBERT, ALBERT, and even influenced GPT models. This ensures that the included studies reflect recent modern advancements in AI, particularly those influenced by large language models.
- **Document Types**
  - *Rationale:* To ensure the quality and reliability of the findings, the search will include only peer-reviewed publications such as journal articles or conference papers. Grey literature, editorials, and non-scientific sources will be excluded.

Based on the keywords above, a search query was derived, which is presented in the appendix A

This search query was applied to `Scopus`, `ACM Digital Library` and `IEEE Xplore`. These libraries offer comprehensive and complementary coverage of relevant literature. Scopus provides a broad, multidisciplinary index that ensures wide visibility of peer-reviewed research. IEEE Xplore is essential for accessing high-impact publications in software engineering and performance testing, particularly from industry-driven and standards-focused perspectives. The ACM Digital Library complements these by offering rich content from leading conferences and journals in software engineering and artificial intelligence, ensuring a well-rounded and in-depth exploration of the topic.

3.3.4 Study Selection and Eligibility Assessment

The search query was adapted to fit the syntax of the selected libraries without altering or removing its semantic meaning. Running the search query on `Scopus` revealed 380 papers, `IEEE Xplore` revealed 48 papers, and `ACM Digital Library` returned 1133 papers. The PRISMA flow diagram presented in figure 3.1 shows step-by-step how these papers were processed to track down papers relevant to this mapping study. Ultimately, 37 papers were considered for this analysis.

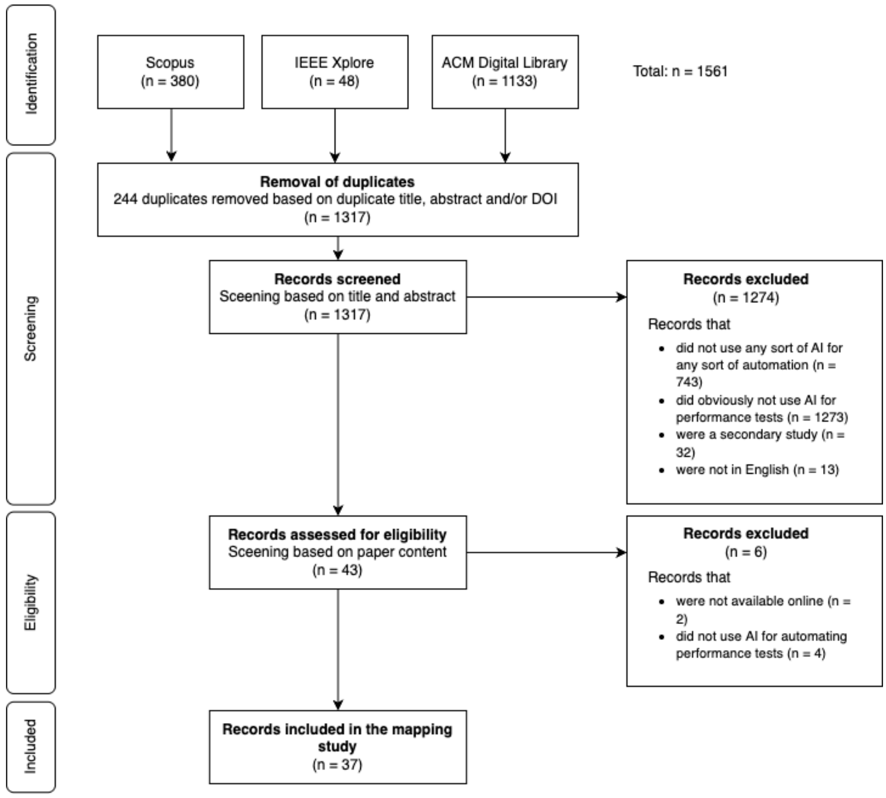

Figure 3.1: Prisma Flow diagram for screening and selection process

3.3.5 Data Extraction

The included papers have been analyzed based on criteria derived from the research questions.

- **RQ1: How did the research field evolve over the past years?**
  - *Publication Year:* Gain insights in how the research area is evolving
  - *Application Domain:* Types of systems that are tested with AI automated performance tests
- **RQ2: What are the existing approaches to automating performance testing in software systems using AI?**
  - *AI Technique Used:* What kind of AI is used in the automation process? E.g. large language models, basic machine learning approaches, deep learning
  - *LLM Model Used:* If an LLM is used, what model was leveraged? E.g. Bert, GPT, Claude
  - *Automation Scope:* Performance testing involves several key steps, such as creating test scenarios, executing selected scenarios and analyzing the performance test results. This metadata aims to clearify the parts of performance tests that got automated using AI.
  - *Integration with Tools:* How were performance tests automated? Did they extend existing performance testing tools or did they develop a custom frameworks?

- **RQ3: How effective are AI-based approaches compared to other performance testing baselines?**
  - *Baseline comparison:* Was a comparison made between traditional and AI based performance testing?
  - *Baseline Type:* To what baseline type were AI-based approaches compared to? E.g., Manual testing, Rule-based, Static analysis
  - *Reported Improvement:* If it was concluded, that AI-based approaches offer benefits, what are they? E.g., Accuracy, Time saved, Coverage
  - *Limitations:* What limitations were reported? E.g., Complexity, Data requirements
- **RQ4: What metrics and methods are used to evaluate the success of AI-driven performance testing?**
  - *Evaluation Method:* How was the evaluation carried out? E.g., Case study, Benchmarking, Simulation
  - *AI Evaluation Metrics:* How did the AI perform? Are there measurable metrics?

The analysis for the listed metadata is based on the whole content of the included papers. This was considered necessary as most of the chosen metadata to analyze is highly specific and unlikely to be included in the abstracts.

3.4 Results

3.4.1 RQ1: How did the research field evolve over the past years?

Figure 3.2 illustrates the temporal evolution of publications in this field. The data reveals a significant growth phase from 2018 to 2020, with publications increasing from a single paper to eight papers. Following this initial surge, the field entered a stabilization phase (2021-2024) characterized by annual fluctuations between four and eight publications. The single publication recorded for 2025 (as of June) is expected to increase by year-end. This pattern, rapid growth followed by stabilization, is typical of emerging research areas, where initial enthusiasm transitions to more focused investigations as the field matures.

The application domains targeted by these studies, depicted in Figure 3.3, provide further insight into the field's evolution. Cloud Computing and Web Applications emerge as primary research focuses, together accounting for approximately 40% of the published works. This emphasis reflects the inherent performance challenges in these domains: Cloud Computing environments introduce complexities such as resource elasticity, multi-tenancy, and distributed architectures, while Web Applications require continuous performance validation to maintain responsiveness across diverse user bases and frequent updates.

The concurrent trends in publication volume and application diversity suggest a field that experienced rapid initial growth, driven by broader AI advancements, and is now developing more specialized research trajectories across multiple application domains. This evolution demonstrates increasing recognition of AI's potential to address complex performance testing challenges across various software systems.

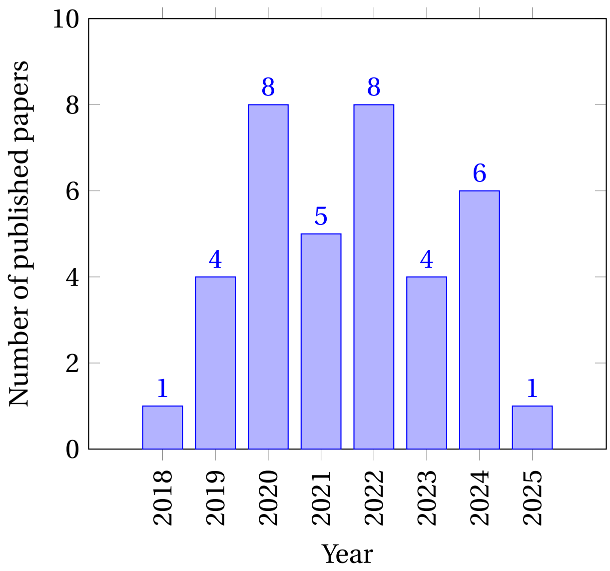

Figure 3.2: Number of published papers over time

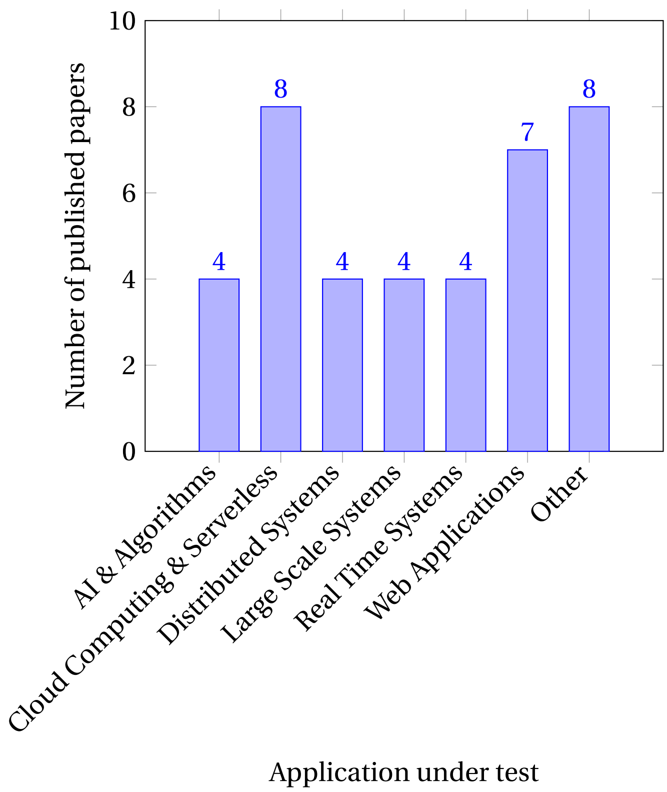

Figure 3.3: Domain of applications under performance test

3.4.2 RQ2: What are the existing approaches to automating performance testing in software systems using AI?

Recent research on automating software performance testing using artificial intelligence highlights a diverse range of approaches, with machine learning emerging as the most prevalent technique featured in 16 studies. Deep learning and reinforcement learning follow, with 6 studies each, while natural language processing (NLP), evolutionary algorithms, and other AI methods are less commonly applied. These techniques are employed across various stages of the performance testing lifecycle. The analysis phase is the most frequently automated, with 29 studies leveraging AI to interpret results, detect anomalies, and predict system behaviour. The execution and preparation phases are also addressed, however, to a lesser extent, with 14 and 11 studies, respectively, focusing on test scenario generation, selecting the right scenarios to execute, and optimising the performance of test executions. This distribution suggests a strong emphasis on data-driven analysis while also indicating a growing interest in achieving end-to-end automation of performance testing through intelligent systems. Furthermore, the use of LLMs and agentic AI has been barely explored for automating performance tests so far, which leaves a huge gap for further research. The papers that do use natural language processing rely on `BART-large`, `Dialogflow` and custom fine-tuned `GPT`-like models. The analysis of AI-driven software performance testing tools reveals a strong prevalence of custom frameworks, often integrated with existing performance testing, monitoring, and orchestration tools such as JMeter, Prometheus, Kubernetes, and CI/CD pipelines. This trend reflects the highly specialized and context-dependent nature of performance testing, where off-the-shelf solutions may not meet the nuanced requirements of diverse systems under test. Integration with established tools like BenchmarkDotNet, JMH, and Chaos Mesh suggests a pragmatic approach, leveraging proven technologies to enhance reliability and reproducibility. Overall, the diversity in tool integration underscores the experimental and evolving nature of this research area, where innovation often requires bridging gaps between AI capabilities and performance engineering practices.

3.4.3 RQ3: How effective are AI-based approaches compared to other performance testing baselines?

Out of the 37 papers included in this mapping study, 26 conducted baseline comparisons with other performance testing methods. Among these, 18 papers referenced conventional manual and traditional approaches. This suggests a significant dependence on established testing techniques as a benchmark for evaluating newer, automated methods.

As shown in Figure 3.6, manual and traditional approaches were the most frequently cited baselines, followed by heuristic/statistical methods (8 papers), machine learning/optimization (7 papers), rule-based approaches (6 papers), and randomized methods (4 papers). Based on the different nature of the used baseline comparison types, a wide variety of improvements and limitations were identified throughout the studies that are presented in Figure 3.1.

3.4.4 RQ4: What metrics and methods are used to evaluate the success of AI-driven performance testing?

To evaluate the success of AI-driven performance testing, researchers have employed a diverse set of methods and metrics, reflecting the multifaceted nature of this emerging field.

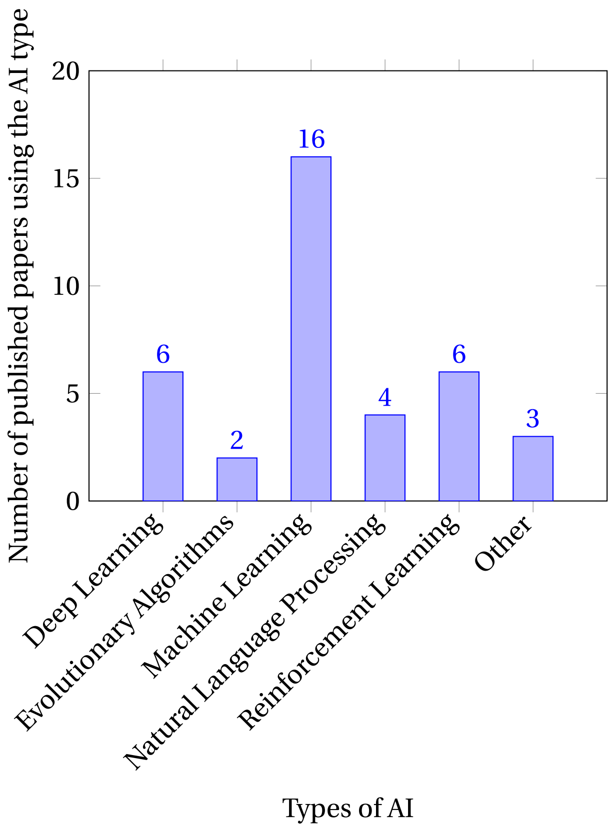

Figure 3.4: Types of AI used for performance test

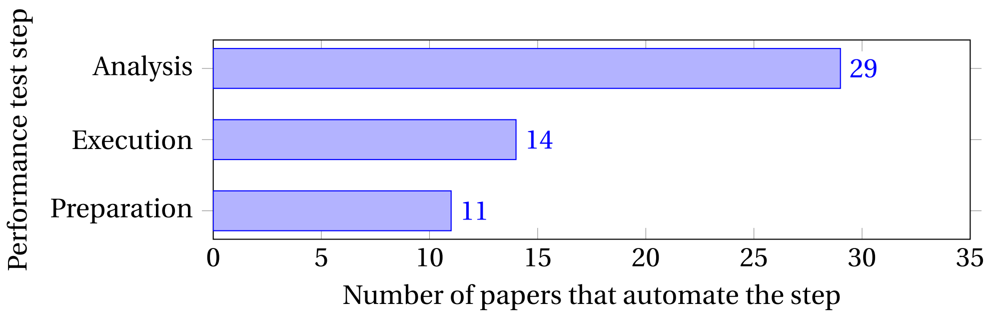

Figure 3.5: Automated performance test steps using AI

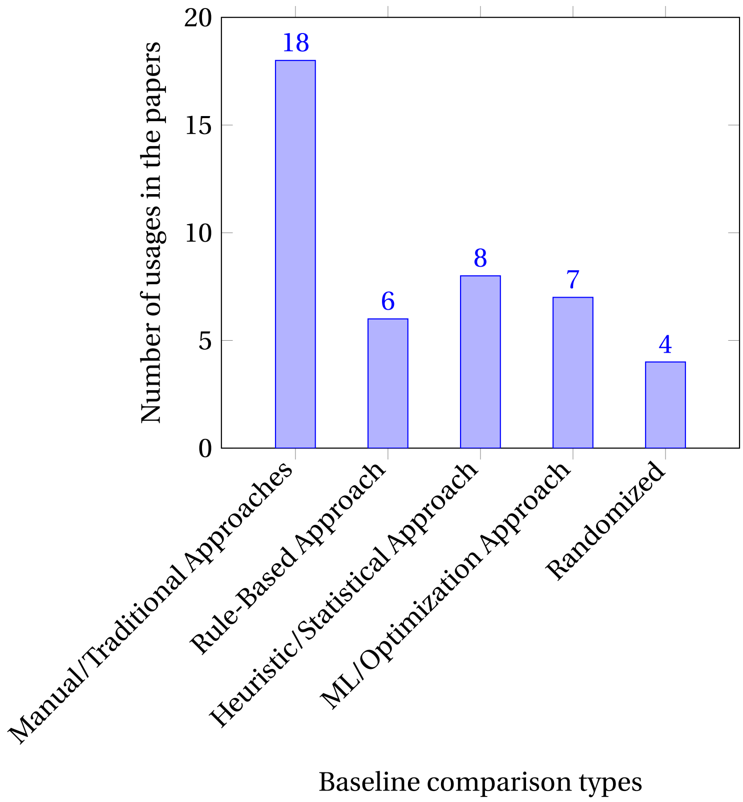

Figure 3.6: Types of baseline comparisons used

Table 3.1: Most prominent improvements and limitations when using AI to automate performance tests

Improvements	**Limitations**
Time saved	Data availability and quality
Accuracy	Scalability of AI models
Cost savings	Model interpretability
Improved reliability	Time-consuming test execution phase
Reduced downtime	Fluctuations in results
Proactive issue resolution	Limited predictor variables
Increased test case effectiveness (IPPstd up to 3.42x)	Hardware Resource constraints for AI models
Proactive issue resolution	Real-time monitoring challenges

Table 3.2: Most prominent AI evaluation metrics

Category	**Metrics**
Classification Metrics	Accuracy, Precision, Recall, F1-score (or F-measure), Specificity, Balanced Accuracy, Confusion Matrix
Regression Metrics	Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), R-squared (R2)
Ranking and Other Metrics	Normalized Discounted Cumulative Gain (NDCG), Mean Average Precision (MAP), Top-N success rate

The most commonly used evaluation method is empirical or experimental evaluation, featured in 28 studies, which underscores the importance of validating AI techniques through controlled and reproducible experiments. Benchmarking is used in 19 studies, highlighting the role of standardised datasets and tools in assessing performance. Case studies (14 studies) and industrial or real-world integration (11 studies) also play a significant role, demonstrating the practical applicability and relevance of AI solutions in real-world settings. Other methods, such as simulation, fault injection or mutation testing, cross-validation, and comparisons with baselines, e.g. simpler AI models, are used less frequently. User or human-based evaluations are rare, suggesting limited exploration of human-in-the-loop or usability aspects.

In terms of metrics, the studies utilise a broad range of evaluation criteria tailored to the nature of the AI task. Classification tasks are assessed using metrics such as accuracy, precision, recall, F1-score, specificity, and confusion matrices. For regression-based evaluations, metrics such as mean squared error (MSE), root mean squared error (RMSE), mean absolute error (MAE), mean absolute percentage error (MAPE), and R-squared (R^2) are commonly used. Additionally, ranking-oriented tasks employ metrics such as normalized discounted cumulative gain (NDCG), mean average precision (MAP), and Top-N success rate. This variety of metrics reflects the huge amount of different and complex approaches to automating performance testing and the need for nuanced evaluation strategies. Overall, the field demonstrates a strong emphasis on empirical rigor and metric diversity, indicating a maturing research landscape focused on both technical accuracy and practical relevance.

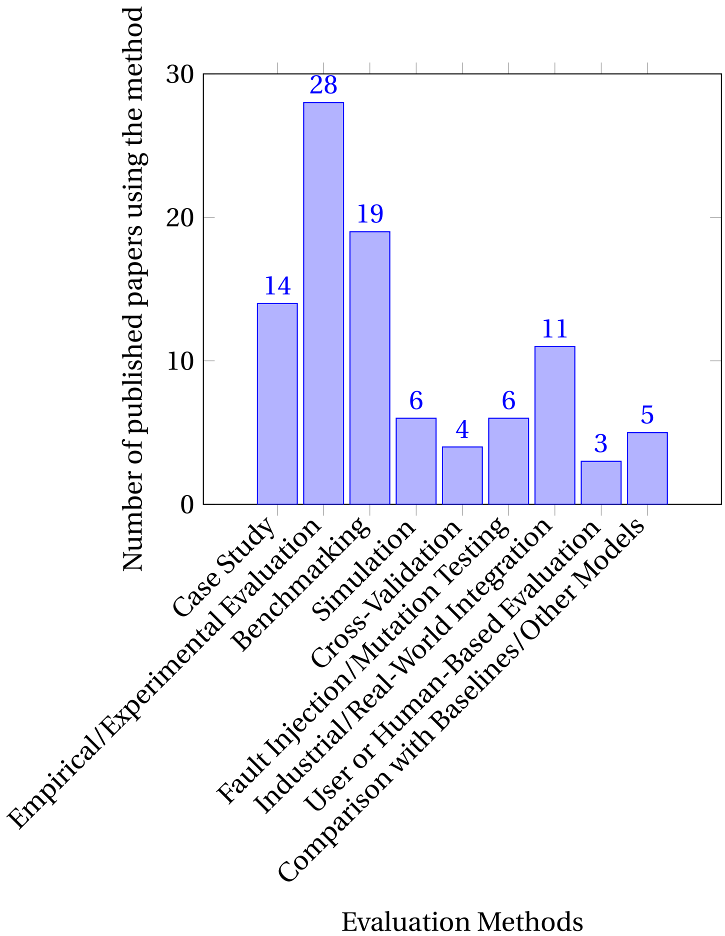

Figure 3.7: Types of Evaluation Methods

3.5 Discussion and conclusion

This systematic mapping study provides a comprehensive overview of the current landscape of AI-driven automation in software performance testing. The findings reveal a rapidly evolving field that has transitioned from exploratory research to studies that are more specialized and application-driven. The initial surge in publications between 2018 and 2020 reflects growing interest in leveraging AI for performance testing, while the subsequent stabilization suggests a maturing research domain with refined focus areas.

Machine learning remains the dominant AI technique, particularly in the analysis phase of performance testing, where it is used to detect anomalies, predict system behavior, and interpret test results. However, the limited use of large language models (LLMs) and agentic AI indicates a significant opportunity for future exploration. The integration of AI into performance testing tools is largely achieved through custom frameworks, often built on top of established platforms like JMeter and Prometheus, highlighting the need for flexible and context-aware solutions.

Effectiveness comparisons show that AI-based approaches often outperform traditional methods in terms of time savings, accuracy, and scalability. Nonetheless, challenges such as data quality, model interpretability, and hardware constraints persist. Evaluation methods are diverse, with empirical studies and benchmarking being the most common, and a wide range of metrics are employed to assess AI performance across classification, regression, and ranking tasks.

In conclusion, while AI has demonstrated substantial potential in automating software performance testing, the field is still in a phase of active development. Future research should focus on expanding the use of LLMs, improving model transparency, and addressing practical deployment challenges. By bridging these gaps, AI can become a cornerstone in the evolution of performance engineering, enabling more efficient, accurate, and adaptive testing processes.

Chapter 4

Problem Solution

In this chapter, a detailed solution to the challenges discussed earlier is presented. Insights from current research and identified needs are utilised. The main ideas, design, and strategy for implementing a new tool that leverages advanced AI to automate software performance testing are explained.

The solution aims to cover each stage of the performance testing process—from creating scenarios to running tests and analysing results—while minimising the need for human involvement. A flexible design is employed, connecting large language models, relevant data sources, and coordination functions to provide a comprehensive automated testing experience.

The following sections outline the reasoning behind key design choices, describe the system's components, and demonstrate how they work together to achieve the established goals.

4.1 Fundamental idea

A desktop application was developed to facilitate end-to-end automation of performance testing processes. The tool enables the management of multiple projects, each encapsulating its own configurations, test scenarios, results, and environments. A guided onboarding experience is provided to assist users in setting up performance tests, including the preparation of environments and the configuration of agents.

Performance tests are structured around specific application features, allowing for targeted and meaningful evaluations. The SUT is not treated as a single, monolithic entity; instead, the different features of the SUT are extracted and evaluated in detail. For each of these features, a full-stack performance test can be conducted, with agentic tools designing and executing tests and collecting relevant metrics.

Test results from various agents and configurations are aggregated and visualised within the application, enabling comparisons across environments such as local, staging, and production.

The solution allows for the updating of configurations and agent setups as the system under test evolves, while preserving historical data for regression analysis. Intermediate steps, such as Dockerization of the application, which is required for PPTAM test execution, and environment setup, are also addressed to ensure seamless integration.

By consolidating all aspects of performance testing into a single interface, the tool presents itself as a performance testing agent that streamlines workflows and enhances the efficiency of performance evaluation efforts.

4.2 Current approach on performance testing

4.2.1 Scenario design

The creation of test scenarios is a critical phase when conducting performance tests, aimed at evaluating how a particular part of the system behaves under specific workloads. The creation of effective performance test scenarios is a nuanced process that requires a deep understanding of the system under test (SUT), its intended usage patterns, and the performance goals of the stakeholders. Therefore, it makes sense to outline the conventional practices followed by performance testers in designing these scenarios, highlighting the sources of information they rely on, the reasoning behind scenario selection, and the methods used to define performance goals.

First of all, a performance tester must investigate the system and moreover the context within it is used. To accomplish that, a comprehensive understanding of the system's architecture, components, and expected behavior must be gathered. Testers typically consult a variety of sources, including:

1. **User stories and requirements documentation**
   - These provide insight into how the system is expected to be used, helping testers identify critical user journeys and usage patterns.
2. **System architecture and design documents**
   - These help in identifying performance-critical components, such as databases, APIs, and third-party services.
3. **Source code and commit history**
   - In some cases, testers inspect the codebase to understand implementation details, identify potential bottlenecks, or detect recent changes that may impact performance.
4. **Stakeholder interviews**
   - Discussions with developers, product owners, and operations teams help clarify performance expectations and identify business-critical scenarios.

After that, the testers define one or several goals depending on the current phase of development and the types of results they want to achieve at the end. There are a number of different types of performance tests as defined by Pargaonkar et. al [9]. The most commonly used are:

- **Load Testing**: Evaluates system behaviour under expected user loads to ensure it meets performance criteria.
- **Stress Testing**: Pushes the system beyond its limits to identify breaking points and assess robustness.

- **Scalability Testing**: Measures the system's ability to scale up (or out) as demand increases.
- **Spike Testing**: Assesses how the system handles sudden surges in traffic.
- **Endurance (Soak) Testing**: Evaluates system stability and resource usage over extended periods.

Once the system context is understood and the goals of the performance test are well defined, testers begin to derive specific test scenarios. This process involves identifying key user interactions, modelling realistic workload, mapping user flows and prioritising scenarios. Based on user stories and analytics data, testers select the most common and performance-sensitive user actions. Within the scope of an e-commerce application, an example for that can be the following order of events:

1. Login
2. Search for a particular product
3. Adding the first product of the results page to the cart
4. Performing the checkout

This results in a step-by-step list of interactions. After obtaining these flows of user interaction, the expected number of concurrent users, request rates, and session durations must be defined based on production data or stakeholder input. This results in a list of end-to-end workflows of fictive users that can then be mapped to simulate real-world usage, often using tools like UML activity diagrams or sequence diagrams. In the end, the testers have a wide variety of scenarios; however, not all of them might provide the same level of usefulness, mainly in regard to how many different features are tested and whether they align with the goals of the performance testers. To compensate for that, the scenarios must be sorted by prioritising the most important ones.

The final step is to implement the performance test scenarios using specialized tools such as JMeter, K6, Gatling, or LoadRunner.

4.2.2 Test execution

Testers must also ensure that the test environment closely mirrors production in terms of hardware, software, and network configurations to obtain reliable results. Therefore, the staging environment is often used for carrying out the tests. However, depending on the chosen goals and the available environments, the development/local environment or the production environment might be better and/or cheaper choices that might still satisfy the required quality of results. For example, in case the goal of the performance test is to uncover memory leaks by using endurance testing, it is likely that this fault also occurs in the development environment, but maybe to a lesser extent. Therefore, it makes sense to execute the performance test locally and try to observe the suspected problems there first before running the tests in staging or production. Especially with the rise of cloud computing, which is often billed per request, such an approach is often more cost-effective and may yield the same or similar results. However, when the goal is to observe the maximum capacity of the system (stress testing), it obviously does not make sense to run these types of tests locally, since the developers' machines are completely unrelated to real traffic. In case the staging environment is an exact replica of the production environment, executing the test there would be the most sensible approach, also because it does not impact customers' experiences on the production environment. If it is not an exact replica, the stress test shall be carried out in production, scheduled when the number of real customers is at its lowest, e.g. during nighttime.

As a conclusion, selecting the most suitable environment is crucial for obtaining the required and unobscured test results.

4.2.3 Result analysis

Analyzing the results of performance tests is a crucial step in understanding whether a system meets its performance requirements and identifying areas for improvement. Therefore, it is necessary to outline the standard practices followed by performance testers in interpreting test outcomes, comparing them against expectations, and deriving actionable insights. This exposes points of attack that can be later investigated for automation.

The analysis process varies depending on the type of performance test conducted, but it generally involves a combination of quantitative metrics evaluation, system behavior observation, and contextual interpretation.

The first important step is to establish a baseline and comparison criteria. This baseline may include current system usage metrics, which give insights into real-world data such as average response times, peak user loads, and resource utilisation from production environments. If the software is developed for a client, it may also contain service-level agreements (SLAs), which are predefined performance targets agreed upon with stakeholders. Furthermore, it may also consider historical performance and usage data or even industry benchmarks, defined by standard performance expectations for similar systems or technologies.

Next, testers typically choose key metrics to focus on. These provide raw data on how the system performed during the test execution and must be interpreted during the result analysis phase. Some examples of these metrics are provided below:

- **Response time**: Time taken to process a request, often broken down by endpoint or transaction type.
- **Throughput**: Number of transactions or requests processed per unit of time.
- **Error rate**: Percentage of failed or erroneous transactions.
- **Resource utilisation**: CPU, memory, disk I/O, and network usage during the test.
- **Latency distribution**: Percentile-based analysis (e.g., 90th, 95th, 99th percentile) to understand worst-case performance.

These metrics are visualised using graphs, histograms, and time-series plots to identify trends, spikes, and anomalies. Furthermore, the types of metrics analysed also depend on the chosen goals. For example, in stress testing, the response time is a good indicator to see when the performance of the SUT starts to degrade. For endurance testing, for example, the resource utilisation provides insights into bottlenecks.

Afterwards, these metrics must be analysed based on the type of performance test conducted. The type already professes the goal, the focus of analysis and the outcome. Some examples are presented below.

- Load Testing

  - *Goal*: Validate system performance under expected user load.
  - *Analysis Focus*: Stability of response times, throughput consistency, and resource usage under normal conditions.
  - *Outcome*: Determines whether the system can handle its intended workload without degradation.

- Stress Testing
  - *Goal*: Identify system limits and failure points.
  - *Analysis Focus*: Behavior under extreme load, error rates, system crashes, and recovery mechanisms.
  - *Outcome*: Reveals bottlenecks and helps plan for capacity upgrades or failover strategies.

- Scalability Testing
  - *Goal*: Assess how performance scales with increased load or resources.
  - *Analysis Focus*: Linear or non-linear growth in throughput and resource usage, efficiency of scaling mechanisms.
  - *Outcome*: Informs architectural decisions and infrastructure planning.

This procedure guides the analysis phase, ensuring it stays focused on the performance aspects for which the test was planned.

Finally, the gathered data is interpreted and reported. This means that beyond raw metrics, testers interpret results in context. This means that metrics are correlated, for example, linking high response times with CPU spikes or memory saturation. Root causes for the observed behaviour are identified, and a final report is written, summarising results in dashboards. These reports can be tailored to technical and non-technical stakeholders, often with recommendations for remediation.

4.3 Identifying attack surfaces for automation

The aim of this solution is not to reinvent the wheel and devise yet another completely different approach to performance testing. Instead, the various steps discussed in section 4.2 will be revised, and a strategy will be developed for each of them on how to approach their automation if possible.

4.3.1 Scenario design

As stated in the previous section, testers must develop a fundamental understanding of the application. This is typically done by inspecting the source code and other documents related to the SUT. To automate this step, it is therefore necessary to provide as much context as possible to the large language model to generate reasonable scenarios. Agentic AI, particularly Langgraph, offers an ideal solution in the form of tools. Tools serve effectively as integration capabilities, enabling the AI to access external resources, such as APIs, web search, or code execution tools, thereby extending its functionality. It allows the AI to invoke predefined programmatic functions and retrieve results autonomously, while also allowing the LLM to select the necessary parameters for invoking these functions. Such functions can either be defined in the agent's source code or made available using Model Context Protocol Servers (MCP-Servers). These are standalone executables that can be started when the agent is started, allowing for communication with the agent using Server-Sent Events or standard input/output. It is a way of encapsulating functionality for improved reusability across various agents and extending the agent's capabilities in a unified manner, which most publicly available LLM models support. Based on the analysis made in section 4.2, the following tools were chosen to satisfy the need for context and, therefore, provide better end results:

1. **Codebase Tools**: This set of tools provides the LLM with read-only access to the source code of the SUT. It allows navigation of the folder structure, reading files, and searching the entire codebase using Regular Expression search queries. Given that mode codebases offer ReadMe files and other documentation, this allows the AI to gather sophisticated knowledge about the purpose of the project while also providing an opportunity to collect implementation-specific insights

2. **GitHub Tools**: These give the LLM additional context regarding issues, planned milestones and a commit history. It enables the AI to track the current progress of development and identify potential difficulties that may affect the project's performance.

3. **Atlassian Jira/Confluence Tools**: Jira is one of the most well-known project management tools in the industry. It allows users to create issues and define user stories. Confluence, in contrast, acts as a knowledge base for the project. It is a place for storing and organising documentation and planning documents. By allowing LLMs to access these documents, the quality of output can be significantly improved.

4. **Swagger/OpenAPI Tools**: Insights into the offered endpoints also provide the LLM with an understanding of the offered features and functionality. On the oneenable the LLM to quickly understand how end-users utilise the SUTf the SUT. On the other hand, it will allow the LLM to understand how end-users use the SUT quickly.

Not all projects utilise all the listed third-party tools; therefore, it was decided that only the Codebase MCP server must be present for the agent developed in this thesis to function. The reason for that is that all software projects typically have a codebase of some sort, so it is logical to benefit from this already existing source. However, connecting also the other listed tools essentially increases the AI's understanding of the project. In that case, if a software company, for example, is already working with Jira, it can also benefit from using it for designing performance test scenarios.

Unfortunately, there is no silver bullet for transitioning from starting at the provided context to achieving functional and well-structured scenarios. However, the process can be divided into x subsequent steps that are universally applicable, and agentic AI can be leveraged to devise a suitable approach for each of these steps, depending on the provided context and the nature of the SUT.

1. **Understanding the project**

    - *Goal:* Obtain a fundamental understanding of the SUT
    - *Solution:* Develop an agent that has access to the provided context and writes a project description for the SUT by determining its purpose. This description can later be used to instruct subsequent agents.

2. **Understanding the technical details**

   - *Goal:* Obtain implementation specific insights
   - *Solution:* Use the codebase tools with a specified agent to obtain insights into the architecture and the used tech-stack. Furthermore, also analyse the authentication mechanisms, if present, from which the test generation agent can later benefit.

3. **Understanding the offered functionality**

   - *Goal:* Get a list of SUT features. Since it is difficult to develop a single performance test for the entire project, targeting performance tests to specific features seems more reasonable and also decouples the process of designing test scenarios from the size of the codebase and System Under Test (SUT).
   - *Solution:* Develop an agent that uses the provided context to generate a list of features of the SUT. Each of these features must be described using a name, a detailed description, the value it provides for end users, and the level of importance (low, medium or high). The generation of test scenarios can then target a selected feature, granting granularity and high coverage.

4. **Understanding the feature's difficulties**

   - *Goal:* Brainstorm about possible performance bottlenecks when interacting with the feature
   - *Solution:* Utilise a dedicated agent that analyses the tech stack and the codebase to understand if performance issues regarding the selected feature exist. This procedure can partially drive scenario creation by pointing out possible problems to the scenario creation agent, thereby obtaining more sophisticated results that have a higher chance of uncovering performance issues.

5. **Generating scenarios for a feature**

   - *Goal:* Generate a locust test file
   - *Solution:* Use the information gathered from the previously listed agents to generate scenarios in the form of a Locust file for the SUT. This also involves providing the agent with the existing context again to determine what API endpoints exist and debugging tools that enable the agent to test individual API calls for the SUT on the fly while writing the Locust file, thus ensuring its validity.

6. **Generating a PPTAM configuration**

   - *Goal:* Generate a PPTAM configuration for executing the Locust file
   - *Solution:* Provide all the already gathered information to an agent and generate a test configuration file for PPTAM, used for the performance test execution.

The suggested approach offers a standardised method for designing scenarios. By utilising agentic AI for each step, this approach ensures it is completely independent from the System Under Test (SUT). Performance testers can intervene after each agent execution using the user interface to adjust the results if needed, thereby eliminating the risk of scenario creation failure or generating unusable results. The orchestration of these agents is therefore handled by the user interface, which starts the next agent as soon as the performance tester has verified the results of the previous agent. Implementation of an agent-to-agent architecture can begin once it is confirmed that performance testers are no longer needed. However, this exceeds the scope of this thesis and will not be discussed further.

4.3.2 Test execution

Automating the execution of performance tests presents a particularly complex challenge due to its strong dependency on the current state of the software under test (SUT). As software evolves over time, through new feature additions, architectural changes, or infrastructure updates, the conditions under which performance tests must be executed also shift. This dynamic nature introduces significant variability in test environments, making full automation of this phase difficult.

Moreover, the choice of execution environment (e.g., local, staging, or production) often depends on the specific performance goals, such as identifying memory leaks, validating scalability, or stress-testing system limits. These decisions typically require contextual awareness of the development lifecycle, deployment infrastructure, and operational constraints—factors that are difficult to generalise or automate reliably.

Given these complexities, fully automating test execution in a way that adapts to ongoing development progress and system evolution is beyond the scope of this thesis. However, one practical approach to mitigate this challenge is to enable the re-execution of previously generated performance tests on demand. This allows testers to validate performance repeatedly as the system changes, without requiring the regeneration of test scenarios each time. Such a mechanism supports iterative testing workflows while maintaining a manageable level of automation.

4.3.3 Result analysis

In traditional performance testing, result analysis is a manual, expertise-driven process that involves interpreting raw metrics such as response times, throughput, and error rates in the context of predefined goals. As described in subsection 4.2.3, testers typically compare these metrics against baselines derived from SLAs, historical data, or industry standards. They then visualize the data using graphs and dashboards, identify anomalies, and attempt to correlate them with potential root causes—often relying on intuition and experience. This process is time-consuming, error-prone, and challenging to scale, particularly in agile environments where rapid iteration is crucial.

To overcome these limitations, it seems feasible to outsource the result analysis using agentic AI. It can autonomously examine raw performance test data, understand the operational profile of the SUT, and contextualise the results based on the specific performance test goal, e.g. load, stress, or endurance testing. The outcome is a structured, developer-friendly report that not only summarises key metrics but also interprets them in the context of the system's expected behaviour, identifies potential bottlenecks, and offers actionable recommendations.

4.4 Overall architecture

In this section, the overall architecture of the developed tool will be presented. Therefore, a detailed overview of the different components and concepts will be given in form of diagrams and textual explanations.

The proposed solution consists of the following major building blocks:

- **GUI**: Tauri[1] with Svelte[2] was used to develop the user interface. Tauri allows developers to create desktop applications using web technologies, such as Svelte and other UI libraries, to minimise development time. Unlike Electron[3], Tauri does not produce gigantic bundle sizes by leveraging the OS's webview components instead of Chromium and using Rust as its backend, which also results in better performance. The ultimate goal is to have a single executable that offers pleasurable performance and bundles all dependencies within it.

- **Langgraph Agent**: The Langgraph agents are all written in Python, packaged using *pyinstaller* into one standalone executable and packaged into the Tauri binary.

- **MCP Servers**: The MCP Servers are written in various languages. Other authors have already developed some of the previously described MCP Servers, whereas others have been developed within the scope of this thesis using Python. All of them have been packaged into standalone executables and also packaged into the Tauri binary.

- **PPTAM Test Runner**: The PPTAM application was wrapped with an HTTP Server to allow for complete external control. As stated previously, PPTAM is used to execute performance tests and collect the raw result data (metrics).

This list provides a first peek into the development of the presented tool. To build on that, these blocks will be analysed in the following subsections, and the relationship between them will be explained in detail.

4.4.1 System context overview

Figure 4.1 presents a first-level overview of the implemented architecture. The performance tester (user) interacts with the *Performance Testing Agent*, which guides the tester through the process of creating test scenarios and then executes these scenarios against the *System Under Test*. During scenario creation, several AI agents are utilised to automate as many steps as possible during the performance testing journey. These agents must rely on a large language model to provide them with reasoning capabilities. Therefore, GPT models from OpenAI and Claude from Anthropic were both considered; however, in terms of performance testing, GPT 4.1 mini yielded the best results while also costing much less than Claude per token. Therefore, all developed agents leverage GPT models. To provide them with sufficient context about the SUT, many MCP servers were utilised, allowing the agents to query information about various aspects of the SUT, as presented in the diagram.

[1] `https://tauri.app/`
[2] `https://svelte.dev/`
[3] `https://www.electronjs.org/`

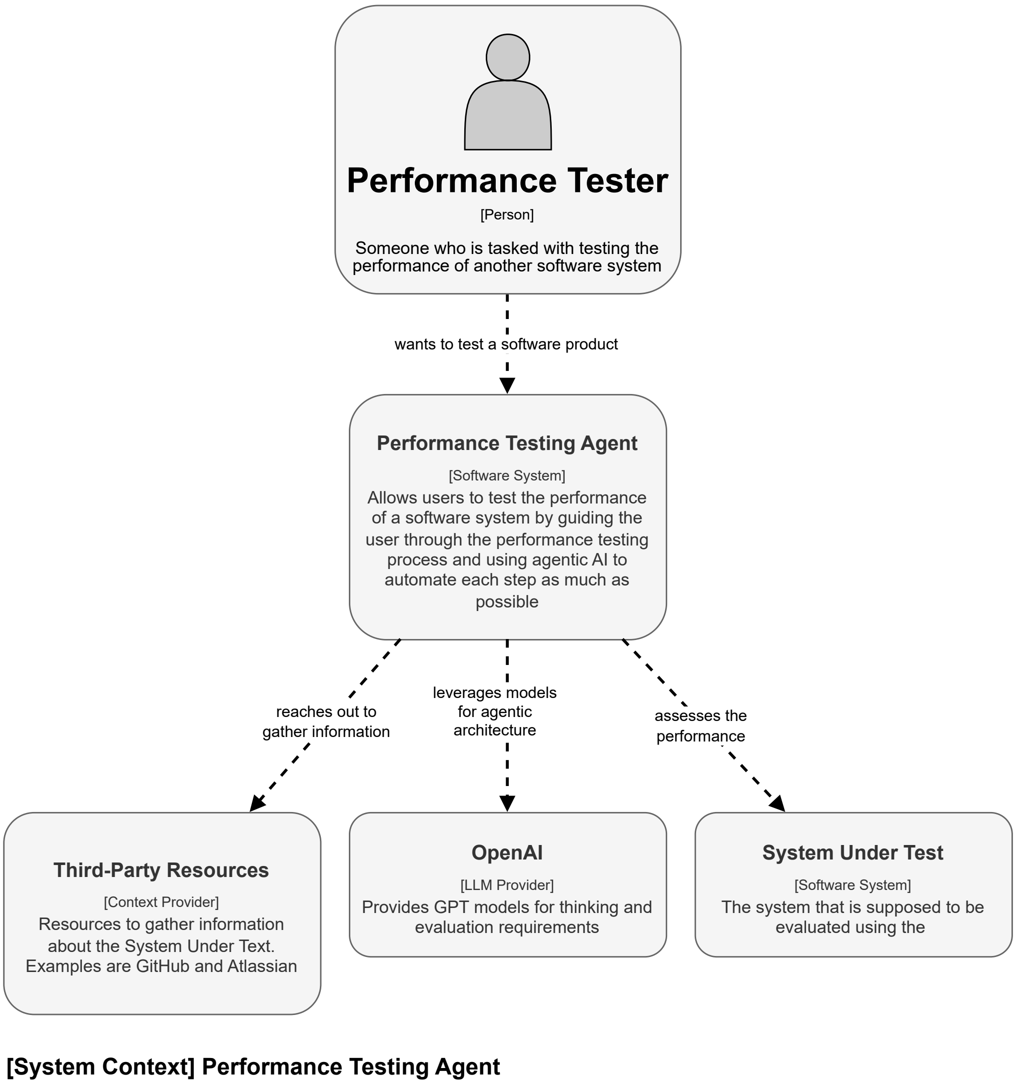

Figure 4.1: System Context Diagram

4.4.2 Container overview

Figure 4.2 illustrates the components of the *Performance Testing Agent* and also embeds the already presented MCP-Servers, pointing out how all these components interact with each other. The *Performance Testing Agent* consists of four major elements: the Tauri Desktop application, a SQLite database, the PPTAM Test Runner, and the bundled Langraph Agents. The Tauri app serves as an orchestrator that coordinates the Langgraph agents, stores their results in the SQLite database and interacts with the *PPTAM Runner* to execute performance tests. The *Langgraph Agent* is a bundled binary that combines the various agents presented in the previous sections. Most of these agents require additional context about the SUT. Therefore, they can leverage the four packaged MCP-Servers to gain the necessary information they need. However, it is worth noting that not all agents always utilise all four MCP servers. Section 4.5 will provide deeper insights and reasons for this design decision.

4.4.3 The Performance Testing Agent Context

This subsection presents detailed component diagrams for all three containers within the *Performance Testing Agent* context: the *Desktop Application*, the *PPTAM Runner*, and the *Langgraph Agent* binary.

Desktop Application Component

Figure 4.3 covers the architecture of the Desktop Application. Built with Tauri v2, the application consists of a frontend developed in Svelte and styled using DaisyUI[4], and a backend that handles core logic and persistence. The frontend communicates with the backend through Tauri's command interface and also interacts directly with the SQLite database via the Tauri SQLite adapter. This adapter not only facilitates local data storage but also enhances the developer experience by allowing secure database access from the frontend code, which is particularly beneficial in this case, as the web application is only accessible within the Tauri WebView.

A key architectural feature of the backend is the use of sidecars, which are external binaries bundled with the application and executed alongside the main Tauri process. Sidecars are employed to manage auxiliary services that are better handled outside the main application runtime. In this project, sidecars are used to launch two local HTTP servers: one acts as a wrapper for the PPTAM module (see *PPTAM Runner*), enabling HTTP-based interaction with its functionality, and the other serves as an interface for communicating with LangGraph agents (see *Langgraph Agent*), which are part of a language model orchestration system. This approach allows the application to maintain modularity and separation of concerns while leveraging native performance and simplified deployment. By offloading these services to sidecars, the application remains lightweight and secure, yet capable of integrating complex external systems seamlessly, which leverage libraries and tools that are incompatible with Rust, like Langgraph and PPTAM.

PPTAM Runner Component

To enable automated performance testing within the PPTAM framework, a containerised component, referred to as the PPTAM Runner, was developed. This component encap-

[4] `https://daisyui.com/`

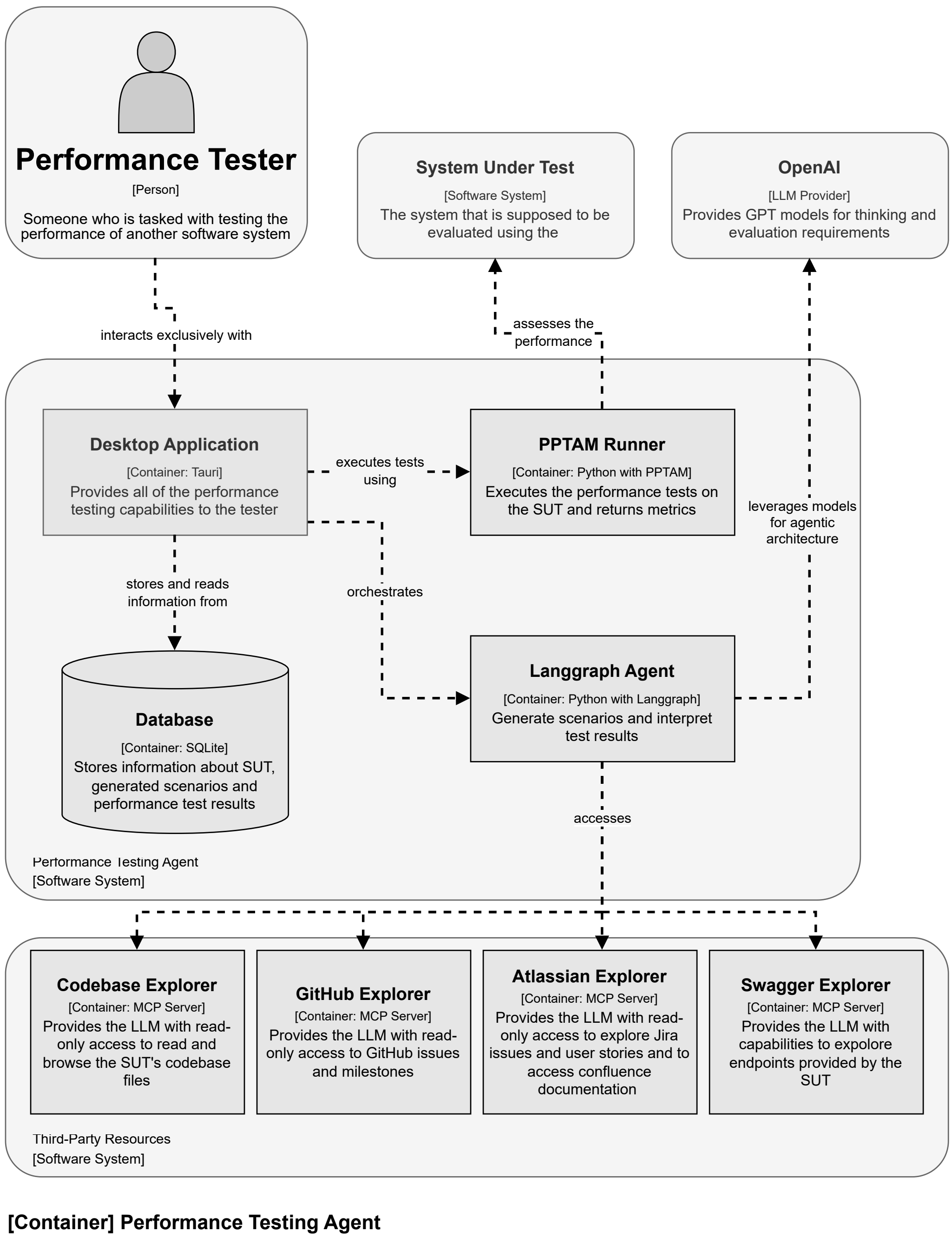

Figure 4.2: Container Diagram

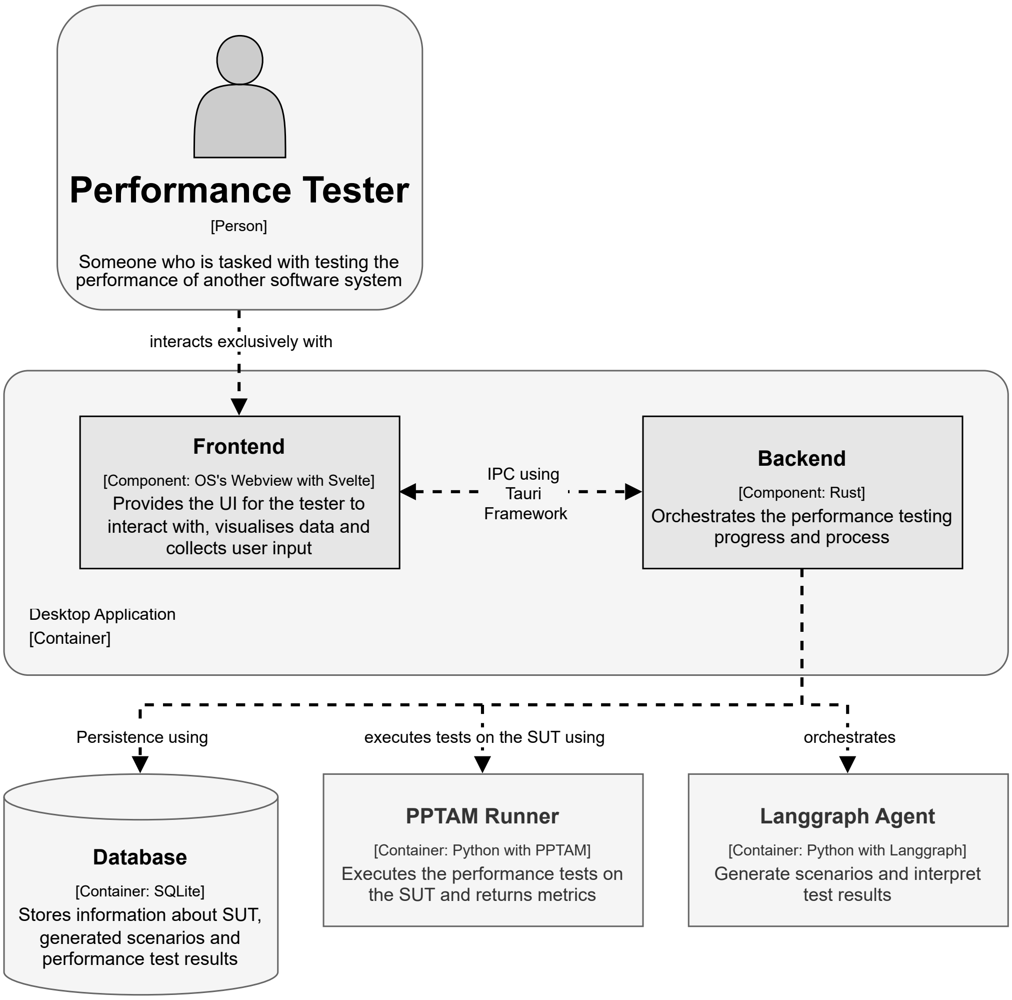

Figure 4.3: Component Diagram - Desktop Application

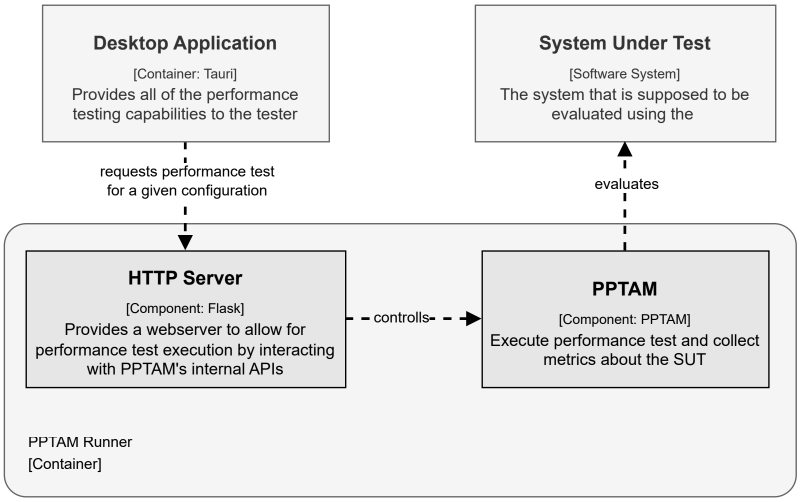

Figure 4.4: Component Diagram - Desktop Application

sulates the core functionality of PPTAM and extends its capabilities through a lightweight HTTP interface implemented using the Flask framework. The interface exposes a RESTful POST endpoint designed to receive test configurations dynamically. Traditionally, PPTAM relies on .ini files located within the SUT directory to define test parameters. However, in automated testing scenarios where different configurations are required based on specific features or testing goals, a more flexible approach is necessary. To address this, the HTTP wrapper was designed to accept configuration data directly in the request body. Upon receiving a request, the configuration is parsed and passed to PPTAM's internal APIs, which are then used to execute the corresponding performance test. Once the test execution is complete, the raw performance metrics representing the test outcome are returned in the HTTP response. This architecture enables dynamic, on-demand test execution as soon as the scenarios are ready. The associated C4 component diagram, shown in Figure 4.4, illustrates the internal structure of the PPTAM Runner, including the interaction between the HTTP interface, the internal PPTAM APIs, and the dynamic configuration mechanism at a high level.

Langgraph Agent Component

Since the Langgraph framework is currently only available for Python and JavaScript, a separate component has been constructed in Python that encapsulates all the AI Agents required for this project. Therefore, a similar approach to the PPTAM component has been taken, where an HTTP Server wraps the core logic of the element, providing endpoints that are invokable from the Tauri backend process to enable bidirectional communication between these two processes. As presented in Figure 4.5, the HTTP Server then invokes one of the nine available agents to respond to the Desktop Application's request. When the Tauri backend calls the endpoint to invoke an agent, it sends along the configuration for the MCP Servers, such as the path to the codebase folder for the *Codebase MCP Server*, as well as results from previous executions from other agents, so that the current agent can build upon those results to avoid re-analysing the SUT over and over again and allowing the agent to just focus on one particular aspect of the SUT. The architectures of the presented agents will be discussed in section 4.5 in detail as it would exceed the overall architecture presentation of the whole tool.

4.4.4 The Third Party Resources Context

In this subsection, the chosen MCP-Servers will be explained in detail, highlighting how the support the agents with fulfilling their tasks.

The Model Context Protocol (MCP) is an open, standardized communication protocol designed to enhance how applications interact with large language models (LLMs) by managing contextual information more efficiently. Developed by Anthropic, MCP addresses the limitations of traditional API-based interactions with LLMs, particularly around context persistence, memory management, and tool integration[19].

At the core of MCP is a client-server architecture. In this setup, MCP servers act as intermediaries between LLMs and external data sources or tools. These servers expose specific capabilities, such as access to files in the codebase of the SUT, APIs to interact with GitHub, Jira or Atlassian, or computational tools, through a standardized interface. This allows LLMs to interact with external systems in a structured, secure, and extensible manner[20].

MCP servers provide three primary types of capabilities:

1. **Tools**: Executable functions that LLMs can invoke (e.g., database queries, API calls).
2. **Resources**: File-like data that can be read or retrieved (e.g., documents, structured data).
3. **Prompts**: Predefined templates that guide the LLM in performing specific tasks.

Unlike traditional LLM API calls, where the full conversation history must be resent with each request, MCP servers maintain persistent context. This enables more natural, multi-turn interactions and reduces token usage by intelligently summarizing and pruning context. As a result, applications built on MCP can support advanced features like retrieval-augmented generation (RAG), tool use, and long-term memory.

MCP is increasingly used in agentic AI systems, where LLMs act as autonomous agents capable of reasoning, retrieving information, and taking actions. By standardizing how context and tools are integrated, MCP servers make it easier to build scalable, modular, and interoperable AI applications.

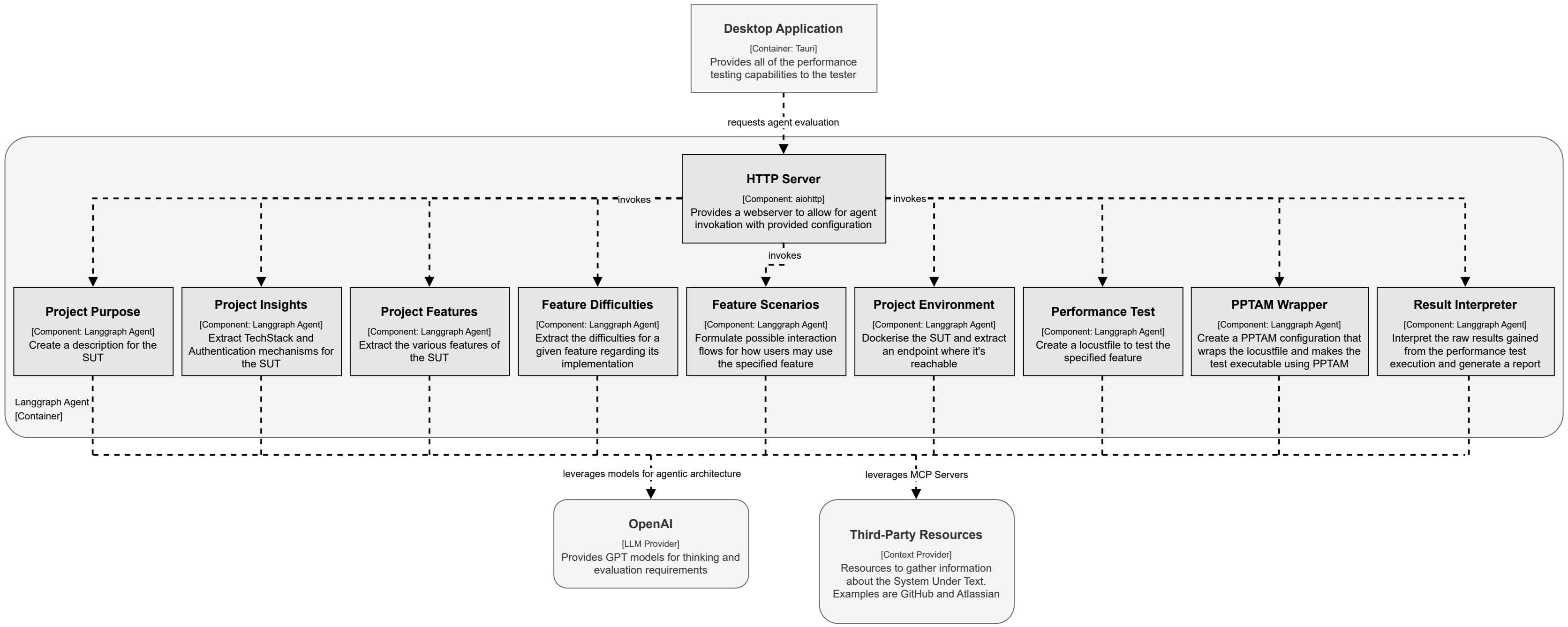

Figure 4.5: Component Diagram - Langgraph Agent

Within the scope of this project, a *Codebase Explorer MCP Server* and a *Swagger Explorer MCP Server* were developed, and other third-party MCP Servers were integrated to provide the LLM with a large amount of sources about the SUT from different perspectives. Those will be explained separately in the following subsections. In perspective of the proposed tool, however, only connecting the *Codebase Explorer MCP Server* was deemed necessary, as all software projects typically have a codebase. All of the other included MCP Servers can be connected optionally, in case a software project already as a Jira board, for example, or its repository can be found on GitHub. All MCP Servers provide the LLM with a different point of view of the SUT, therefore, it is evident that the more MCP Servers are connected, the better the contextual understanding of the LLM is.

Codebase Explorer MCP Server

The *Codebase Explorer MCP Server* provides read-only access for the LLM to retrieve files from the codebase of the System Under Test (SUT). While open-source developers have developed several MCP Servers, it has become clear that the functionality they offer to the LLM exceeds what the agents for this project actually require. During testing of these Codebase MCP Servers, such as `https://github.com/DeDeveloper23/codebase-mcp`, it was observed that the LLM became overwhelmed by the range of available functionalities. As a result, it struggled to identify essential files, such as documentation markdowns, and to navigate the overall folder structure effectively. To compensate for that, a custom codebase MCP server was developed by observing how the LLM works with the codebase and experimenting with the offered functionality. For example, after providing basic functionality, such as *get_file_content(path)* or *list_files_in_directory(path)*, it became evident that the LLM also needed a search function for the entire codebase to find information about features implemented in different parts of the codebase. Therefore, a *find_files_with_content(regex)* function was added. By testing this new function with the LLM, it was observed that it was heavily utilised, especially within codebases that feature a microservice architecture, where parts of a feature are implemented in different services.

The following list hints the entire functionality exposed to the LLM:

- **list_files_in_directory(path)**: Lists all files and directories in the given path. This includes hidden files and works non-recursively.
- **list_files_in_root_directory()**: Lists all files and directories in the root directory. This includes hidden files and works non-recursively. It is a wrapper around *list_files_in_directory("/")* to make it easier to call.
- **get_file_content(path)**: Reads the content of a file. Consider using *get_specific_file_content(path, regex)* to get an overview of the file and query it for specific information.
- **get_specific_file_content(path, regex)**: If the file is a source-code file, it will query the file for the given regex. The LLM can suggest a regex, e.g. to list all functions or classes in the file. This is useful to get an overview of the file without reading the entire content. The regex should be a string, e.g. `def\s+\w+\s*\(.*\):` to match all function definitions in Python files.
- **find_files_with_content(regex_or_str)**: Lists all file paths in the codebase where their content contains the given regex or string. The content should be a string or a regex,

e.g. "translation|i18n" to match all occurrences of the string "translation" or "i18n" in the codebase.

The configuration for this MCP Server consists of the absolute path to the SUT's codebase folder, provided by the Tauri backend process when a request is made to invoke an agent.

Swagger Explorer MCP Server

The *Swagger Expolorer MCP Server* exposes the SUT's API documentation in the Swagger/OpenAPI format allowing the AI agent to programmatically retrieve a comprehensive overview of the available endpoints, their parameters, expected responses, and usage semantics. This structured access to the SUT's interface enables the agent to reason about the system's functionality, e.g. for extracting the offered features, and automatically generate locust test files tailored to the API's behavior without needed to extract those endpoints from the codebase or other documents. The agent that profits the most from this MCP Server is the *Performance Test* agent. By leveraging this documentation, the agent constructs a fully functional Locust test file without having to figure out how the SUT's endpoints are used.

To avoid overwhelming the LLM by simply providing the `swagger.json` file as a whole, the following abstractions were introduced, which enable the LLM to retrieve the required information selectively.

- **list_endpoints()**: Lists all available endpoints in the server together with their invocation method, such as `GET` or `PUT`.
- **describe_endpoint(path, method)**: Returns the description of a specific endpoint when invoked using one of the available HTTP methods, such as `GET` or `PUT`
- **list_endpoint_parameters(path, method)**: Returns the query parameters of a specific endpoint.
- **describe_endpoint_security(path, method)**: Returns the security requirements of a specific endpoint, i.e. if and how the user must be authenticated.
- **list_endpoint_response_codes(path, method)**: Returns the response schema of a specific endpoint.
- **describe_endpoint_response(path, method, response_code)**: Returns the response schema of a specific endpoint for a specific response code, e.g. `200` or `404`
- **find_schema_models()**: Returns the ORM model names used in the server.
- **find_schema_model(model_name)**: Returns the fields of a specific ORM model.

The configuration of this MCP Server differs significantly from that of the other MCP Servers presented in this section. Because the Swagger endpoint that exposes the definition (`swagger.json`) is only accessible when the SUT is running, it would mean that the SUT must always be reachable when any of the presented agents operate. To accommodate this, the *Desktop Application* asks the user for an endpoint where the `swagger.json` file can be accessed when configuring this MCP Server for the first time and then caches this definition file in the SQLite database, thus eliminating the need for the SUT to be always running. By applying this concept and providing the user with the possibility to update or re-download the `swagger.json` file when needed, the nature of this MCP Server is therefore static rather than dynamic. As a consequence, the Tauri backend must also submit the cached swagger definition every time this MCP is used.

GitHub MCP Server

The GitHub MCP Server (`https://github.com/github/github-mcp-server`) is a powerful implementation of the Model Context Protocol (MCP) that enables seamless integration between LLMs and GitHub's extensive API ecosystem. Designed to enhance the capabilities of AI agents, the server serves as a secure, structured interface through which LLMs can perform a wide range of GitHub-related tasks, including querying repositories, managing issues and pull requests, interacting with GitHub Actions, and even conducting code security scans.

By exposing GitHub's functionality through a standardised protocol, the MCP server allows LLMs to gain deep contextual understanding of the system under test (SUT). For example, an LLM can retrieve repository metadata, analyse commit histories, inspect CI/CD workflows, and examine open issues or pull requests. This contextual awareness enables the model to reason about the structure, behaviour, and performance characteristics of the SUT in a way that would be difficult through static code analysis alone.

This integration is particularly valuable for generating automated performance tests. The LLM can use the MCP server to identify performance-critical paths in the codebase, such as frequently modified files, hot paths in workflows, or components with high issue density, and then generate targeted performance tests, similar to the *Codebase Explorer MCP Server*, just with less optimized exposed functionality when it comes to understanding a project. It can also monitor CI/CD pipelines to detect regressions or bottlenecks and suggest optimisations based on historical trends. Additionally, the server supports toolset customisation, allowing developers to enable only the relevant GitHub APIs for a given task, which helps streamline the LLM's decision-making and reduces unnecessary context.

In terms of configuration, the performance tester must create a GitHub API Key with read-only permissions for the targetted SUT's repository and insert that key when asked in the frontend. When sending the HTTP request to the *Langgraph Agent* HTTP Server, this API key is also attached to the request body.

Atlassian MCP Server

The goal of this MCP Server is to extend the capabilities of the LLM by offering structured, real-time access to project management and documentation data. This integration enables an LLM-based agent to interact with Jira and Confluence through a standardized protocol, allowing it to query, analyze, and act upon the information stored in these platforms.

From a SUT perspective, this integration is invaluable. Jira provides a rich source of structured data about the development lifecycle, such as issue types, priorities, statuses, user stories, and sprint progress, while Confluence offers unstructured but highly contextual documentation, including design specs, meeting notes, and architectural decisions. By accessing both, the LLM can build a multi-dimensional understanding of the SUT, encompassing not just the codebase, the CI/CD results, but also the intent, planning, and execution context behind it.

For example, using this MCP Server, the agent can:

1. Analyze Jira tickets to identify recurring performance-related issues or bottlenecks.

2. Correlate Confluence documentation with code changes to understand the rationale behind architectural decisions.

3. Generate performance test cases based on historical bug reports, epics, or non- functional requirements that were documented.

4. Summarize sprint retrospectives or planning documents to infer areas of technical debt or performance risk.

Moreover, the MCP server supports natural language commands, enabling the agent to perform tasks like "Find all unresolved performance bugs in the last two sprints" or "Summarize the scalability section of the architecture doc." This empowers the LLM to act as a context-aware assistant that can proactively extract features and understand the direction the project is going to in the future.

For configuration, the performance tester must create an Atlassian API Key and fill out the required information, including the workspace of the SUT. Similar to the *GitHub* MCP Server, this information is sent along when submitting an HTTP request to the *Langgraph Agent* HTTP Server.

4.5 Agent architectures

LangGraph builds upon the capabilities of LangChain, a framework designed to simplify the development of applications powered by LLMs. LangChain provides abstractions for managing prompts, memory, call chains, and integrations with external tools and APIs, including the previously mentioned MCP Servers. It enables developers to construct complex LLM-driven workflows by chaining together modular components such as language models, retrievers, and tools in a linear or branching sequence, thus allowing for a more dynamic interaction with LLMs.

LangGraph extends this paradigm by introducing a graph-based execution model, allowing for more flexible and dynamic control flows. While LangChain is well-suited for relatively straightforward pipelines, LangGraph is designed for stateful, multi-agent systems where the flow of execution may depend on intermediate results, agent decisions, or external signals. In contrast to workflows, where the order of steps is well defined in advance, agents can build their own plan to solve a specific problem. This is of exceptional value for the proposed tool as the quality of the codebase of the SUT may vary a lot and depending on the connected MCP Servers, the contextual understanding of every aspect of the SUT cannot be assured, therefore the AI requires much more freedom in how it interacts with the provided resources. In LangGraph, each node in the graph represents a computational unit, often an agent or function, and edges define the possible transitions between these units based on the evolving state.

This architecture is particularly advantageous for automating performance testing, where tasks such as test generation, execution, monitoring, and analysis must be coordinated in a non-linear, adaptive manner depending on the SUT. LangGraph enables agents to operate autonomously while maintaining shared context, supporting asynchronous execution, and allowing for conditional branching and iterative refinement. By building on LangChain's foundational abstractions and extending them with graph-based orchestration, LangGraph revealed itself as a highly suitable approach for constructing intelligent agents that automate performance tests while handling complex workflows with minimal human oversight.

4.5.1 Fundamental agent architecture

To avoid reinventing the wheel, the architecture for all implemented agents relies on a modified version of the `ReAct` architecture. This will be illustrated below, along with the modifications made to it to ensure it operates within the scope of performance testing.

The ReAct architecture

The `ReAct` architecture consists of four nodes: a `START` and an `END` node, a `Thinking` node and a `Tool` node as shown in Figure 4.6. It is a framework designed to enable agentic AI systems to perform complex reasoning and decision-making by interleaving two core capabilities: reasoning and acting. In this paradigm, agents are structured to alternate between generating internal thoughts (reasoning) and invoking external tools or functions (acting), thereby allowing them to iteratively refine their understanding of a task and take informed actions. This approach has proven effective in scenarios that require both cognitive planning and interaction with external systems or environments.

LangGraph extends the `ReAct` paradigm by introducing a graph-based execution model, which comes pre-compiled with LangGraph and in which the agent's behaviour is represented as a directed graph. Each node in the graph corresponds to a specific phase in the agent's reasoning (`Thinking` node) or action (`Tool` node) process, and a shared, mutable state governs transitions between nodes. This structure allows for flexible, stateful workflows that can adapt dynamically based on intermediate results.

The typical state of the `ReAct` architecture looks like this:

- **messages**: List<Message>

The `START` node initialises the agent's state by creating an empty array for the messages list and determines the first transition. The `Thinking` node is responsible for internal deliberation, where the agent evaluates the current context, interprets previous outputs gained from the messages list in the state, and decides on the next step. If an external action is required, control is passed to the `Tool` node, where a specific tool or function is executed. The output of this tool is then incorporated into the state, added to the end of the messages list, and the agent returns to the `Thinking` node for further reasoning. This loop continues until the agent determines that it has completed the task, at which point the `END` node is reached and the final output is generated.

The state plays a central role in this architecture. It is a structured data object that persists across node transitions and accumulates all relevant information, including the initial input, a history of thoughts and actions, tool outputs, and any intermediate data required for decision-making. This persistent memory enables the agent to maintain context, adapt its strategy based on prior outcomes, and ensure coherence throughout the reasoning process.

Although the `ReAct` architecture provides a robust foundation for agentic workflows, modifications are often introduced to tailor it to specific domains or tasks, such as performance testing. A modified version of this architecture is employed in the present work and will be described in detail in the following section.

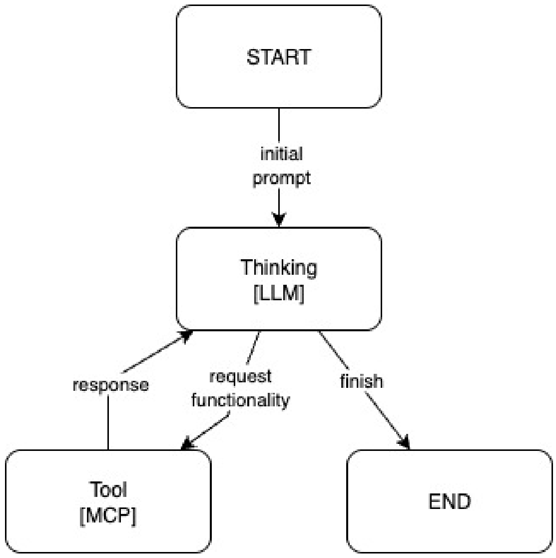

Figure 4.6: ReAct Agent Architecture

Modifications to the ReAct architecture

The core concept of the `ReAct` architecture works perfectly for all the agents implemented within this project. However, some improvements are necessary to enhance the output quality and to address the undefined nature of the SUT, as all agents must be capable of performing their tasks even if some of the configured MCP Servers return insights of poor quality or with a low level of detail.

First, the `ReAct` architecture was modified to limit the number of executions of the `Thinking` node. During implementation, the agents were tested with real projects to ensure that they function correctly. It was observed that agents built on the `ReAct` architecture tend to over-analyse the SUT. For example, when the agents used the *Codebase Explorer MCP Server*, they tended to read every single file in the codebase for specific projects to gain the most comprehensive understanding possible. However, this behaviour is completely infeasible for the analysis tasks they are supposed to carry out. By limiting the number of invocations for the `Thinking` node and providing the LLM with the information on how many calls are still allowed, the LLM was forced to consider which files are worth reading and which files should be skipped. As a result, the agent execution times were significantly reduced, and the output quality was substantially higher, most likely because the context window (the list of messages stored in the graph's state) was less cluttered with irrelevant information. A downside of this approach is finding a suitable threshold for the limit for each agent. This was achieved by testing different limits on different sizes of SUT codebases.

Next, the state of the architecture was expanded again by introducing a new field, called `next_step`. Given that the data quality of the MCP Servers is unknown and that the SUTs can differ significantly in terms of architecture, domain, and complexity, the agent must develop a plan on its own to achieve the desired output. It must compare the quality of each connected MCP Server and find ways to combine information from different sources in a way that allows it to still reason with the data and draw logical conclusions. During development, it was observed that the agent often gets lost in details, thus being incapable of completing the required task. To prevent that, the agent was forced to plan its steps ahead so that it would not miss the overall goal of its execution. Every time the `Thinking` node is invoked, the agent is required to respond with a JSON that allows it to override its next planned step. In addition, the prompt that the LLM is given every time always contains the current value of the `next_step` field in the state, together with the number of remaining executions available for the `Thinking` node and a repeated explanation of the expected results. By consistently adding this "reminder" to the messages array and thereby pushing it into the LLM's context window repeatedly, it can effectively plan its steps based on the remaining execution limit, with the overall goal in mind. The consequence is that the agent is much more likely to complete its task within the given limit, and its actions are much more thought-through. This change also allowed for the usage of the *GPT-4.1-mini* model for the SUT analysis tasks instead of the regular *GPT-4.1* model, which provides a more cost-effective solution and given the larger input and output token limits of the *GPT-4.1-mini* model, the results turned out to be even better than using the regular *GPT-4.1* model, mainly because the analysis phase is all about gathering and combining already present information from the MCP Servers.

Furthermore, the state was extended to feature a `done` field. This allows the agent to break out of the thinking loop as soon as it deems the task to be finished and sets the `done` flag to true. In addition to the already described termination condition, i.e., when the limit of executions of the `Thinking` node is reached, this allows the agent to return early, thus saving tokens and preventing the results from worsening if the agent is forced to continue until the limit is reached.

Lastly, the router that handles the edges between the `Thinking` node and the `Reasoning` node was changed. In the `ReAct` architecture, every call to the `Thinking` node must be followed by a call to the `Tool` node. However, given the previously applied changes, it now makes sense to remove the necessity for a `Tool` node call between two `Thinking` node calls. Since the LLM can now plan its next step and store it within the graph's state, another edge can be added to the graph that allows for transitioning from the `Thinking` node to the `Thinking` node again, effectively creating a smaller cycle. It must be noted that the execution limit will still be reduced by one even when the LLM chooses to skip the `Tool` node call in between.

These changes result in the following enhanced state of the modified architecture:

- **messages**: List<Message>
- **thinking_limit**: number
- **done**: boolean
- **next_step**: string

The final graph of the modified architecture is presented in Figure 4.7, which serves as a reference for all implemented agents.

4.5.2 Agents overview

An overview of the designed agents was already given in section 4.4.3. However, in this section, the interaction between these agents will be explained, focusing on how data is passed between them by laying out the input and output data of each agent. For better visualisation, an enumerated list is presented that highlights the execution order step by step.

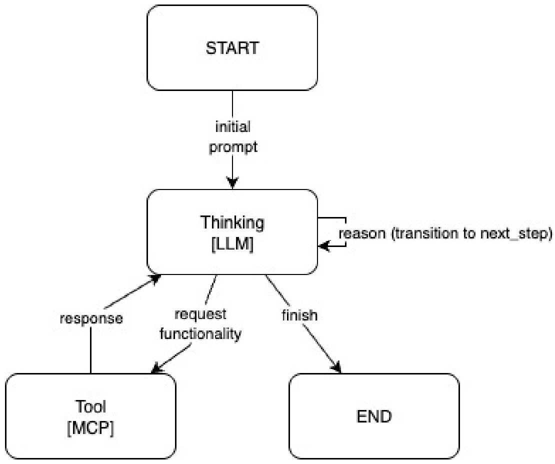

Figure 4.7: Modified Agent Architecture

1. Initial configuration
   - The performance tester is asked to describe the SUT in 1-2 sentences to provide a starting point for the first agent to work with. In addition, the tester configures the *Codebase Explorer MCP Server* and other MCP Servers of choice
2. Invocation of the **Project Purpose** agent
   - *Input*: The 1-2 sentences provided in step 1
   - *Output*: A detailed paragraph explaining what purpose the SUT serves (a detailed description)
3. Invocation of the **Project Insights** agent
   - *Input*: The project's name and the result gained in step 2
   - *Output*: The project's tech-stack and information about the project's authentication mechanisms, if present, like username and password or phone number
4. Invocation of the **Project Features** agent
   - *Input*: The project's name and the result gained in step 2
   - *Output*: A list of the project's features
5. Human input
   - The performance tester selects a feature which will be the focus of the generated performance test. The user can select another feature later to create additional performance tests that focus on other parts of the SUT
6. Invocation of the **Feature Difficulties** agent
   - *Input*: The project's name, the result gained in step 2 and details about the feature under test (FUT) gained in step 4
   - *Output*: A list of possible difficulties that may cause performance issues when users interact with the FUT

7. Invocation of the **Feature Scenarios** agent
    - *Input*: The project's name, the result gained in step 2, details about the FUT gained in step 4 and a list of possible difficulties gained in step 6
    - *Output*: A list of fictive scenarios that each describe one way in which a user may interact with the FUT. Given that the list of difficulties is also provided, the agent will prefer scenarios that particularly involve triggering such a difficulty
8. Invocation of the **Project Environment** agent
    - *Input*: The project's name, the result gained in step 2 and the results gained in step 3
    - *Output*: A `docker-compose.yml` file and one or more *Dockerfile*s depending on the SUT's architecture. This effectively dockerises the SUT to make it compatible with PPTAM. Additionally, an endpoint is extracted where the project's web server will be accessible.
9. Human input
    - The *Project Environment* agent only sets up the local environment. If the tester wants to carry out a performance test on the staging or production environment, those environments must also be configured; however, no agent is provided in this case to automate the process.
10. Human input
    - The performance tester selects a goal for the performance test, such as `Load Testing`, `Stress Testing`, `Endurance Testing`, etc.
11. Invocation of the **Performance Test** agent
    - *Input*: The project's name, the result gained in step 2, information about the FUT gained in step 4, a list of difficulties gained in step 6, a list of sample scenarios gained in step 7, technical insights gained in step 3, information about the goal of the performance test gained in step 10 and an endpoint where the SUT is reachable (step 8 or step 9)
    - *Output*: A Python locust file that implements the performance test
12. Invocation of the **PPTAM Wrapper** agent
    - *Input*: The project's name, the result gained in step 2, information about the FUT gained in step 4, the locust file generated in step 11 and information about the goal of the performance test gained in step 10
    - *Output*: A configuration for PPTAM that describes how it should handle the execution of the performance test for the SUT
13. Human input
    - The performance tester can review the gathered results so far and make modifications as necessary. Then, the generated performance test is executed by clicking on the `Execute` button

14. Background process
    - The generated performance test is sent to the PPTAM HTTP server and executed. The raw results/metrics are sent back to the Tauri application and stored in the SQLite database
15. Human input
    - The performance tester inputs the operational profile of the SUT must fulfil
16. Invocation of the **Result Interpreter** agent
    - *Input*: The project's name, the result gained in step 2, information about the FUT gained in step 4, the defined operational profile gained in step 15 and the raw performance test results gained in step 14
    - *Output*: A report describing the results of the performance test

4.5.3 Agents implementation

This subsection discusses the implementation of the individual agents with respect to the fundamental, modified `ReAct` architecture.

Project purpose agent

This agent takes as input the name of the project and one or two sentences of description provided by the performance tester. This brief description will already guide the agent in examining the project as a whole without delving into particular details, which was discovered during implementation to be a problem when no starting point for the agent was given. In terms of architecture, the graph of the modified `ReAct` architecture shown in Figure 4.7 was applied. However, the state of this agent was extended to allow the agent to refine the project's purpose with each iteration. This method ensures that a valid result is always available even when the execution limit is hit.

The updated state looks like this:

- **messages**: List<Message>
- **thinking_limit**: number
- **done**: boolean
- **next_step**: string
- **purpose**: string

Note that the agent leverages all configured MCP Servers to gain information about the project's purpose. Every time the `Thinking` node is executed, it returns a JSON that updates the `next_step`, `done`, and `purpose` fields in the state if these fields are present in the returned JSON. As soon as the agent finishes, the purpose value from the state is returned.

Project insights agent

The task of this agent is to examine the technical details of the project, specifically the tech stack and authentication mechanisms. These results are of high interest for the *Project Environment* agent and the *Performance Test* agent. By understanding the tech stack first, the accuracy of the later-generated Docker configuration was observed to be significantly higher. Additionally, the authentication mechanisms guide the creation of the locust file to ensure that endpoints requiring authentication are called correctly. Since many projects leverage already existing authentication frameworks, the order of analysis of these two properties is essential. By analysing the tech stack first, possible authentication frameworks can already be identified and analysed more deeply in a second inspection step to understand how they are configured. Given this dependency, the agent's architecture was changed to resemble this restriction. Effectively, two modified `ReAct` agents were sequentially connected, where the first one focuses on understanding the tech stack and the second one focuses on understanding the authentication mechanisms. Figure 4.8 displays the architectural graph of this agent.

The state used for this agent is identical to the one used for the modified `ReAct` architecture. However, since this agent must fulfil two almost independent tasks, two additional state initialiser nodes were inserted. When called, these update the state by setting the `done` flag to false, setting `next_step` to an empty string and setting `thinking_limit` to a value suitable for each of the two analysis phases. Unlike in the *Project Purpose* agent, where an additional field was introduced to keep track of the current state, this agent uses long-term memory tools to keep track of its progress. Since the tech stack and the authentication mechanisms are structured data of possibly large volume that must be collected from across the codebase, likely requiring the agent to investigate deeply into source code files, the risk of cluttering the context window of the LLM would be extremely high using the approach described in the previous agent. Having a message inserted every time the `Thinking` node is executed with all values of the fields in the state unnecessarily fills up the AI's context window. Therefore, it makes much more sense to give the LLM the capability to read and write those insights into an external data structure managed using the long-term memory approach. Effectively, it can be seen as providing the LLM with a CRUD api to manage its state. Thus, whenever the current state tends to leave the LLM's context window, it can decide on its own to re-read the current state, thereby pushing it again into its context window.

Those long-term memory functions are added to the already existing functions provided by all connected MCP Servers. They are represented by the `Tool` node in the respective agent's architecture graph. Since both `Thinking` nodes have access to the same tools, the `Thinking` node in the authentication analysis phase can also access the results of the `Thinking` node of the tech stack analysis.

The exposed long-term memory functions are:

- *get_techstack()*: Returns the current tech stack of the project as a comma-separated string
- *add_to_techstack(item)*: Adds an item to the tech stack of the project
- *remove_from_techstack(item)*: Removes an item from the tech stack of the project
- *get_authentication()*: Returns the current authentication description of the project

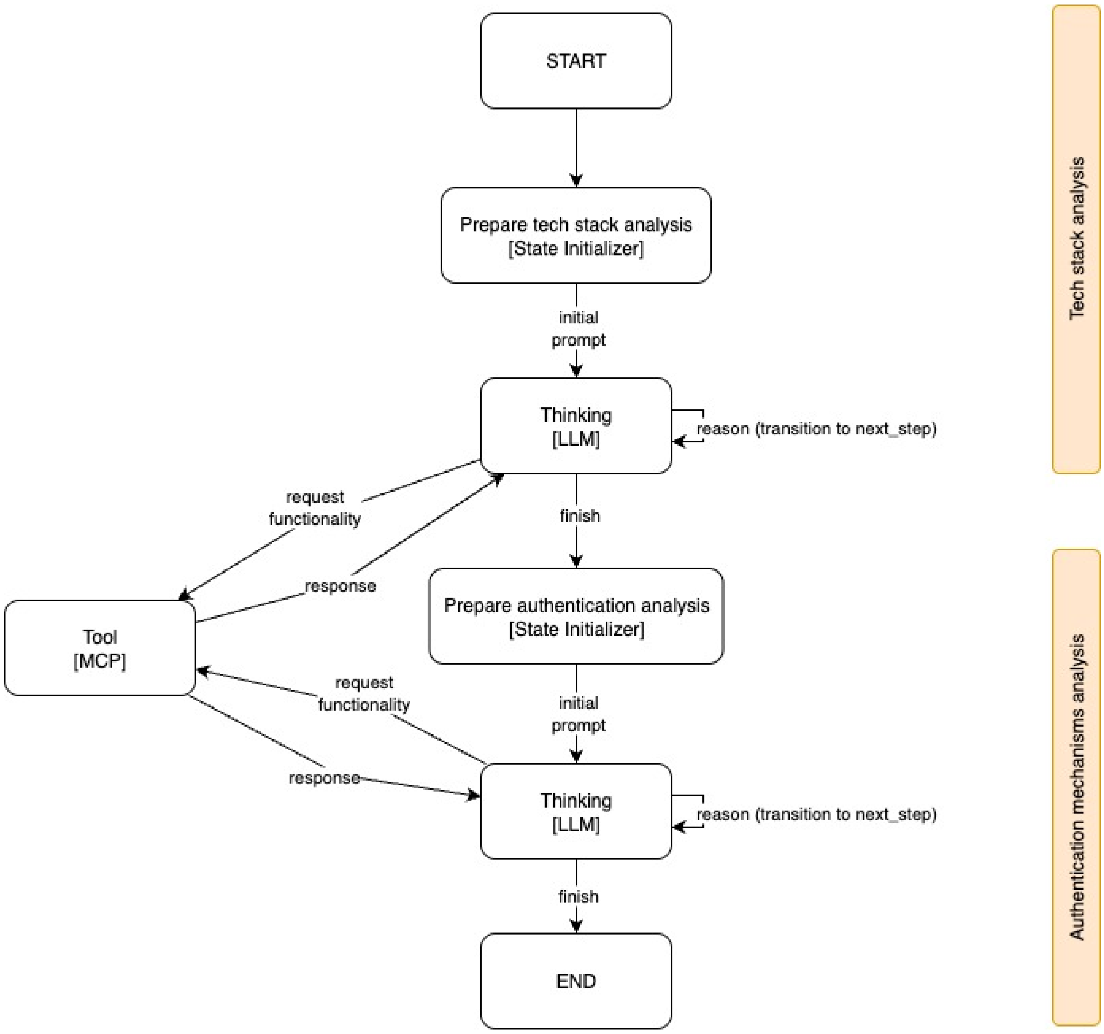

Figure 4.8: Project Insights Agent Architecture

- *set_authentication(method)*: Sets the authentication method for the project. A string that explains how the project implements authentication is passed
- *get_authentication_schema()*: Returns the current authentication schema of the project. A comma-separated list of what fields are needed to authenticate a user, e.g. username, password, phonenumber, ...
- *set_authentication_schema(fields)*: Sets the authentication schema for the project. A list of required fields is passed that a user needs to authenticate

Project features agent

The *Project Features* agent is executed to extract the various features implemented by the SUT from a domain perspective. The goal is to identify independently testable features by creating a list with the following properties for every feature:

1. *name*: The name of the feature.

2. *description*: A brief description of the feature.

3. *value*: The value that this feature provides to the user when using it.

4. *importance*: The importance of this feature for the project. Acceptable values are "high", "medium" or "low".

5. *references*: A list of references related to this feature, e.g. files in the code or JIRA tickets.

6. *icon*: An emoji to represent the feature, must be a single emoji, used for visualisation purposes in the frontend.

The architecture and the state for this agent are identical to the modified `ReAct` architecture shown in Figure 4.7. However, just like in the *Project Insights* agent, long-term memory tools were appended to the tools provided by all connected MCP Servers to allow the agent to modify the list of features during the thinking phase.

The long-term memory tools are composed of

- *get_features()*: Returns the names of all features as a comma-separated string
- *get_feature_by_name(name)*: Returns the properties of a specific feature identified by its name.
- *add_feature(name, description, value, importance, references, icon)*: Adds a feature to the long-term memory
- *remove_feature(name)*: Removes a feature identified by its name
- *update_feature(name, description, value, importance, references, icon*: Updates a feature's properties identified by its name. Only the fields requested to be updated must be provided; `None` can be used to skip updating a specific property, thus allowing the LLM to concentrate on one specific aspect of a feature.

These functions are optimised to be used by the AI to continuously refine the feature analysis while exploring the context provided by the MCP Servers. It provides a way for the LLM to query its memory in a fine-grained manner, keeping the impact on the context window as minimal as possible, while also granting write access, which allows for modification of individual properties to minimise confusion with the values of other properties.

Feature difficulties agent

This agent aims to identify potential issues with the implementation of a given feature that may lead to performance issues. This analysis serves as a baseline for the *Feature Scenarios* agent, guiding it to create scenarios that are more likely to uncover bottlenecks and identify problems with the SUT's resource usage. It applies the modified `ReAct` architecture presented in Figure 4.7, together with its state. Moreover, the agent has access to the *Codebase Explorer MCP Server* to identify implementation issues, as well as all additional connected MCP Servers, to identify conceptual problems with a feature. Like the previous agent, this agent also leverages the use of long-term memory tools to manage its analysis progress step by step using the following CRUD functionality:

- *get_difficulties()*: Get the names of all difficulties
- *get_difficulty_by_name(name)*: Get a difficulty's description by its name
- *add_difficulty(name, description)*: Add a difficulty to the list of difficulties
- *remove_difficulty(name)*: Remove a difficulty by its name
- *update_difficulty(name, description)*: Update a difficulty's description by its name.

Feature scenarios agent

To generate a locust file that covers the entire flow of how a user interacts with a feature, it is crucial first to consider how a fictitious user might interact with the given feature. To accommodate this, the *Feature Scenarios* agent was developed to generate fictitious user flows that describe, step by step, the actions a user performs during the usage of the specified feature. The modified *ReAct* architecture was therefore enhanced using long-term memory tools to allow the agent to work on the scenarios while exploring the feature iteratively.

The long-term memory consists of the following functions:

- *get_scenario_names()*: Returns a list of the names of all scenarios
- *get_scenario_by_name(name)*: Returns the scenario with the given name
- *add_scenario(name, user, description)*: Adds a new scenario with the given name, user, and description. The user is represented by a sample name of a user performing the described scenario. The description is a list of events that the user performs in this scenario, one after another.
- *update_scenario(name, user, description)*: Updates an existing scenario with the given name. Only the provided fields that are not `None` are updated
- *delete_scenario(name)*: Deletes the scenario with the given name

Project environment agent

The goal for this agent is to dockerize the SUT to make it compatible with PPTAM. PPTAM uses a `measure_docker_stats` plugin to gain insights into execution metrics of the SUT, which requires the SUT to be runnable with `Docker`. The architecture of this agent employs the modified `ReAct` architecture with its state and has only access to the `Codebase Explorer MCP Server` in combination with long-term memory tools to work on the build files required by Docker (`Dockerfiles`) and the `docker-compose.yml`. Additionally, the endpoint is extracted where the SUT's web server will be accessible locally when executed using Docker.

The LLM can interact with the long-term memory using the following API:

- *get_dockerfiles()*: Return the paths of all Dockerfiles in the project.
- *get_dockerfile(path)*: Returns the current Dockerfile at the given path. If the file already exists, it will return its content; otherwise, it will check if a Dockerfile for the given path has already been created and return that one instead. Alternatively, an error message is returned

- *set_dockerfile(path, content)*: Sets the Dockerfile content for a specific Dockerfile at a given path
- *get_docker_compose()*: Returns the current Docker Compose file content for the project
- *set_docker_compose(content)*: Sets the Docker Compose file content for the project
- *get_base_endpoint()*: Returns the base endpoint where the project is hosted locally, e.g. `http://localhost:8000`
- *set_base_endpoint(endpoint)*: Set the base endpoint where the project is hosted locally

Performance Test agent

PPTAM relies on a `Locust file` for execution. A `Locust file` is a Python script that defines how virtual users behave during a load test using the Locust framework, which PPTAM uses under the hood. It describes the actions users take, such as visiting web pages, submitting forms, or interacting with APIs, and how often they perform these actions. The file includes user classes that simulate different types of users, tasks that represent specific behaviours, and optional wait times to mimic realistic user pacing. This setup enables testers to simulate real-world usage patterns and measure a system's performance under load.

This agent is tasked with creating such a `Locust file`, primarily by considering the previously generated scenarios, information about the feature under test, the goal of the performance test (e.g., `Stress Testing`), and technical insights, such as authentication mechanisms. This agent, like most of the other agents, is also fully implemented using the modified `ReAct` architecture. However, unlike other agents, this agent not only uses long-term memory tools for managing its output, but also leverages debugging and knowledge shelf tools.

To verify that a given endpoint works as described in the documentation or `Swagger` definition, the agent was equipped with a debugging tool, allowing it to perform REST API calls independently, thus the agent can ensure that the generated `Locust file` is compatible with the exposed REST API of the SUT. As a consequence, the SUT must be running when this agent is executed, which is ensured by the `Desktop Application`.

This debugging tool was exposed to the LLM in the following was:

- *def call_api_endpoint(endpoint, method, headers, body, timeout)*: Call an API endpoint with specific request data, such as the request method (e.g. GET or POST), the HTTP headers, a specified request body if applicable and a timeout in seconds.

Since the agent takes as input most of the results of previous agents, it is infeasible to pass these results to the AI within the initial or thinking prompt due to the large volume of data. As a solution, knowledge-shelf tools were developed that function similarly to long-term memory tools without exposing their functionality to the AI, allowing it to override those values. The LLM can query the results from previous agents on demand and utilise long-term memory tools to write the `Locust file` simultaneously.

The knowledge-shelf tools provide structured access to the following information, utilising an interface already presented in previous agents; however, without the setter or updater functions:

- Scenarios

- Tech stack
- Authentication mechanisms
- Performance test goal
- Feature under test
- Endpoint where the SUT is accessible

The long-term memory tools are the following:

- *get_locust_file()*: Returns the current locust file for performance testing the project
- *set_locust_file(file_content)*: Sets the locust file for performance testing the project

The final `Locust file` is returned after the agent execution terminates.

PPTAM wrapper agent

The purpose of this agent is to define the execution environment in which the locust file will be executed using PPTAM. PPTAM traditionally requires a `configuration.ini` file for setup. To make this process more dynamic and to allow the `Desktop Application` to start a PPTAM test execution dynamically for various generated performance tests, it was wrapped with an HTTP Server that can accept a configuration within the request body. Therefore, the task of this agent is to generate such a configuration based on the `Locust file` and the defined goal of the performance test. The modified `ReAct` architecture was employed in combination with long-term memory tools that allow the agent to create such a configuration. The long-term memory tools expose the following functionality:

- *get_test_plan()*: Returns the test plan for the PPTAM Load Testing Tool in a readable format as a string
- *add_step(step, load, spawn_rate_per_second, run_time_in_seconds)*: Adds a test to the test plan at the given index (step) or the end if step is `None`.
- *remove_step(step)*: Removes a step from the test plan

When the performance test is executed, the generated test plan, along with the project's name, endpoint, and directory path, is sent to the PPTAM HTTP Server, which then runs the performance test according to the generated plan.

Result interpreter agent

The last step in the execution flow is to interpret the performance test results. Therefore, the *Result interpreter* agent was developed. It takes as input the already generated information about the project, the feature under test, the goal of the performance test, as well as the operational profile and the raw performance test results. The ultimate goal of this agent is to write a report that evaluates the results of the executed performance test and places them in context with the specified operational profile. To do that, this agent also relies on the modified `ReAct` architecture and leverages its state. Just like explained for other agents, this agent has its suite of long-term memory tools, knowledge-shelf tools and math tools.

To work on the report, the agent uses the following long-term memory tools:

- *get_report_sections()*: Return the titles of all sections in the report.
- *get_report_section_by_title(title)*: Return the content of a section by its title. This allows the report to concentrate on one particular section during generation
- *add_section_to_report(title, content, position)*: Add a section to the report. If `position` is not `None`, the new section can be inserted between other sections
- *update_report_section(title, content)*: Update an existing section in the report
- *delete_section_from_report(title)*: Delete a section from the report
- *get_report()*: It was observed that the agent can benefit from reading the entire report at once, especially when it decides to perform a quality analysis

To allow the agent to access the raw test results and the operational profile at any time while looping over the `Thinking` node, that data is made accessible with knowledge-shelf tools.

During the implementation of this agent, it was observed that the agent struggles with performing mathematical assessments, particularly when comparing raw test results with values defined in the operational profile and when drawing conclusions from the raw test results. To accommodate this weakness, the agent was provided with a tool to perform Python operations during execution. This allows it to perform any mathematical evaluation using the Python interpreter.

4.6 Solutions mapped to problems

This section provides a structured mapping between the requirements defined in Section 2.2 and the corresponding elements of the developed solution described in this chapter. The goal is to demonstrate how each requirement, whether goal-oriented, domain-specific, product-related, or design-driven, has been addressed through specific architectural decisions, agent implementations, or user interface features.

4.6.1 Requirements-Solution Mapping

Table 4.1 presents a comprehensive mapping of all requirements to the respective solution components. Each requirement is categorised according to the schema introduced in Section 2.2 and linked to the part of the system that fulfils it.

4.6.2 Tactics–Solution Mapping

In addition to requirement fulfilment, the system design also incorporates several architectural and design tactics to ensure usability and robustness. Table 4.2 outlines how these tactics are realised in the solution.

Table 4.1: Requirements-to-Solution Mapping

Type	Requirement	Mapped Solution Component
Goal	Developer wants to avoid performance issues	End-to-end automation via agentic AI; minimal manual intervention
Domain	Developer wants to avoid spending time on performance tests	Scenario generation, execution, and analysis, fully automated by agents
	Developer wants clear instructions to fix issues	Result Interpreter Agent generates actionable reports
	Customer wants quick resolution of performance issues	Continuous re-execution of tests; fast feedback loop
	Customer wants issues fixed before deployment	Integration with local/staging/production environments via PPTAM
Product	AI-based solution for test generation, execution, and analysis	LangGraph agents and MCP servers orchestrated via Tauri UI
	Configurable solution for different software types	Modular MCP server architecture; user-configurable endpoints
	Understandable AI behaviour	Modified ReAct architecture with transparent reasoning steps
	Obtainable result reports	Result Interpreter Agent outputs structured, human-readable reports
Design	Implemented in Python	All agents and MCP servers developed in Python
	Minimal user input; use existing artefacts	Codebase, Swagger, GitHub, Jira MCP servers provide context
	Use LangGraph for scenario generation	All agents implemented using LangGraph
	Use PPTAM for test execution	PPTAM wrapped in HTTP server and integrated into Tauri app
	Generate reports with performance issues	Result Interpreter Agent produces detailed performance reports

Table 4.2: Tactics–Solution Mapping

Tactic	Purpose	Implemented In
Automation	Reduce manual effort in performance testing	Agentic AI for scenario generation, execution, and analysis
Modularity	Enable extensibility and maintainability	MCP server architecture; separate agents for each task
Human-in-the-loop	Allow user intervention at key stages	Tauri UI prompts user to review agent outputs
Context-awareness	Improve AI reasoning with rich context	Integration with codebase, Swagger, GitHub, Jira via MCP
Transparency	Make AI decisions understandable	Modified ReAct architecture with next_step and thinking_limit
Reusability	Enable repeated test execution	Re-execution of tests without regenerating scenarios
Scalability	Support different system sizes and architectures	Architecture-independent design; Docker-based deployment

Chapter 5

Evaluation

This chapter describes the evaluation of the developed proposed solution and, in particular, the implemented AI agents using a descriptive, scenario-based approach. Therefore, the developed tool will be executed on two suitable real-world open-source applications.

5.1 Evaluation objectives

To comprehensively assess the capabilities of the developed tool, the evaluation focuses on several key aspects that reflect both its technical performance and agentic behaviour. Specifically, the objectives are to evaluate:

1. **Outputs of all implemented agents**: Assessing the relevance, diversity, and completeness of the outputs of the implemented agents
2. **Autonomy and Decision-Making**: Analysing the agents' ability to independently select meaningful sources where to find the required information, their ability to reason about them and how they adapt their behaviour based on the structure and characteristics of the target application.
3. **Scalability**: Evaluating how well the tool performs across applications of varying complexity, from small codebases to large-scale systems, and its ability to maintain effectiveness under increased workload.

In addition to standard evaluation, fault injection techniques and controlled modifications will be applied to the selected codebases. These modifications will introduce misleading or ambiguous information, such as altered documentation or additional folders for microservices, without changing the actual behaviour of the system under test (SUT). This will enable an assessment of how well the agents can handle imperfect or deceptive input, as often found in the industry, e.g., when documentation becomes outdated, while maintaining robustness in their decision-making and scenario generation processes.

5.2 Repository selection

To ensure a realistic and meaningful evaluation of the developed tool, two open-source microservice-based applications were selected from the `DeathStarBench` [1] benchmark suite.

[1] `https://github.com/delimitrou/DeathStarBench`

DeathStarBench is a widely recognised collection of real-world, cloud-native applications designed to evaluate the performance of distributed systems. The selected repositories were chosen based on their architectural complexity, relevance to performance testing, and alignment with the evaluation objectives outlined in the previous section.

The following criteria guided the selection process:

1. **Microservice Architecture**: The repositories should consist of multiple interacting services to test the tool's ability to generate complex, multi-step performance scenarios.

2. **Realistic Workloads**: The applications should simulate real-world user behaviour and data flows.

3. **Scalability**: The codebases should vary in size and complexity to assess the tool's scalability.

4. **Modifiability**: The repositories should allow for controlled modifications and fault injection without altering the core logic of the system under test (SUT). Therefore, documentation must be present that can be changed.

Based on these criteria, the following two repositories were selected:

1. Scenario 1 - **Hotel Reservation**:

   - This application simulates a hotel reservation system that handles ten services, including user authentication, room availability, payment processing, and reservation management. It provides a well-defined REST API and realistic user workflows, making it suitable for evaluating the tool's ability to generate meaningful performance test scenarios and handle service dependencies.

2. Scenario 2 - **Social Network**:

   - The social network application models a modern social media platform, including services for user profiles, timelines, media storage, and social interactions (e.g., likes, follows, posts). Its larger scale and more complex service interactions make it ideal for testing the tool's scalability and the agentic AI's decision-making capabilities.

While both applications are built on microservice architectures and represent realistic cloud-native workloads, they differ significantly in structure and complexity, which is valuable for a comprehensive evaluation. The hotel reservation system is relatively minor and linear in its service interactions, making it suitable for evaluating the tool's baseline performance and scenario generation in a controlled environment. In contrast, the social network application features a more intricate and interconnected service graph, with asynchronous communication patterns and higher concurrency. This complexity challenges the tool's scalability, the agents' ability to reason about indirect dependencies and possible performance bottlenecks, and the robustness of scenario generation under more chaotic conditions. These differences allow for a comparative analysis of how the tool performs across varying levels of system complexity and interaction density.

5.3 Experimental setup

To ensure a fair and consistent evaluation across both selected systems, the experimental setup was carefully designed to minimise variability and enhance comparability. The tool was configured to operate in a controlled environment, using the same contextual resources and similar configurations for both the hotel reservation and social network applications.

In particular, both projects were configured to use only the `Codebase Explorer MCP Server` as the sole context provider. This decision was made to isolate the evaluation to the agents' ability to reason and act based solely on static code analysis, without the influence of runtime data or external documentation. Since both projects can be found within the same `DeathStar` repository on GitHub, alongside other applications, it was deemed infeasible to connect the `GitHub MCP Server`. Additionally, none of the selected SUTs expose a Swagger endpoint or utilise Atlassian tools for project management; therefore, the respective MCP Servers were also not connected.

By standardising the context source, the evaluation can more accurately reflect differences in agent behaviour depending on the SUT, since all agents for both projects only have access to one view of the project, namely the codebase.

This setup allows for a direct comparison of how the tool performs across two distinct microservice architectures.

Furthermore, all Docker-related files were removed to test the agents' ability to reconstruct them solely based on the documentation and source code.

The following subsections outline how the developed tool was configured for each project individually.

5.3.1 Hotel Reservation

Two faults were injected into this project's source code:

1. Modifications to the `README.md`

    - The readme of this project lists the supported actions of the project, which are three in total: Get profile and rates of nearby hotels available during given time periods, recommend hotels based on user-provided metrics and place reservations. For the sake of testing the contextual understanding of the agents, one more non-existent feature was added to the list: Hotels in the system can be deleted or added by any user

2. Modifications to the folder structure

    - The `services` directory houses the implementation of all microservices. Another folder was added, called `chat`. The purpose is to mislead the agents into thinking that an eleventh microservice exists, offering a feature that is not described anywhere in the documentation.

These modifications enable an understanding of the depth of analysis performed by the various agents and to realise whether the agents prefer documentation inside the repository or source code for inspecting the SUT. It also reveals whether the agents double-check the information they collect.

Furthermore, the performance tester must provide one to two sentences of description for the SUT, as described in the implementation sections, to provide a starting point for the first agent. To avoid obfuscating the results and retain comparability between both selected projects, a minimal one-liner was provided: *The project is an implementation of a Hotel Reservation system.*

5.3.2 Social Network

The faults injected into this project attempt to mimic industry use cases, where documentation is often sparse or not available at all. Therefore, the following fault was injected:

1. Modifications to the `README.md`
    - This project also features a supported actions section in the readme, just like the other project. However, instead of adding an invalid feature, this section was deleted from the file, together with setup instructions and architectural decisions. Just a paragraph describing the project was left in the file.

The README file is the only documentation within the codebase; therefore, no further modifications to other files were necessary.

Regarding the starting point definition, a similar attempt was taken as described for the `Hotel Reservation` system. The following description was provided: *The project is an implementation of a Social Network system.*

5.4 Execution and collected results

This section provides a detailed description of how the tool was executed with both SUTs in mind, revealing the results gained for each agent execution. Each SUT will be executed separately, generating a performance test that will be executed and evaluated by the proposed tool. In terms of fault injection, the results of every agent will not be corrected after their execution to understand how the tool as a whole reacts to the injected faults.

5.4.1 Hotel Reservation

The results of all agent executions will be laid out and analysed in terms of how they dealt with the injected faults, as well as the SUT in general.

Project Purpose agent execution

After configuring the tool with the specification described in section 5.3, the next step is to create a detailed project description using the *Project Purpose* agent. During execution, the agent stores its thinking steps which are presented in Table 5.1.

This table outlines the sequential reasoning steps taken by an AI agent tasked with generating a comprehensive project description from a codebase. The agent begins by examining high-level documentation, such as the README.md, to establish an initial understanding of the project's purpose and scope. It then systematically explores the structure of the codebase, focusing on the services directory, which houses the core microservices of the hotel reservation system. By diving into specific service directories, namely recommendation, reservation, and search, the agent identifies key implementation files, such as `server.go`, and analyses their contents to understand the functionality and interdependencies of each component. This iterative exploration enables the agent to develop a comprehensive understanding of the system, ranging from its general architecture to the detailed service logic. Finally, it revisits the documentation to refine and validate its interpretation, ensuring its accuracy.

The agent returned the following result as a description for the SUT:

> This project is a comprehensive hotel reservation system implemented as a microservices architecture using Go and gRPC. It provides users with the ability to search for hotels available during specific time periods, receive personalized hotel recommendations based on various criteria such as proximity, rating, and price, and place reservations. The system supports dynamic hotel management, allowing users to add or remove hotels. It integrates multiple services including user profiles, hotel ratings, reviews, geographic data, and chat functionalities to enhance the overall booking experience. The reservation service efficiently manages availability checks and booking processes using caching and persistent storage. The recommendation service offers tailored hotel suggestions to improve user decision-making. Overall, the project aims to deliver a flexible, scalable, and efficient platform for hotel booking and recommendations, enhancing user experience and operational management for both customers and hotel providers.

The agent clearly failed in handling the fault injections. In its generated description, it suggests that a user can add or remove hotels from the system. Additionally, the agent claims that a chat service exists, which is also a result of an injected fault.

Project Insights agent execution

The thinking steps of this agent are shown in Table 5.2. The Project Insights agent follows a structured and methodical approach to analyse a codebase, identifying the technology stack and uncovering authentication mechanisms. It begins by examining foundational files, such as `go.mod`, to determine the core technologies used in the project, identifying Go as the primary programming language, along with gRPC, MongoDB, Consul, Jaeger, and Memcached. The agent then investigates the presence of containerization and orchestration tools by checking for directories like helm-chart and kubernetes, confirming the use of Helm and Kubernetes. Further insights are gathered from the README.md file, which reveals additional components such as Docker, Docker Compose, and OpenShift. To ensure completeness, the agent explores the services directory to detect any other frameworks or technologies used in the microservices implementations. Once the tech stack is deemed comprehensive, the agent shifts focus to authentication mechanisms as presented in the implementation sections of this agent. It identifies that the system employs username and password authentication with SHA-256 hashing and verifies this through the user service proto files. To ensure no other mechanisms are in place, it examines other service directories and middleware for signs of token-based authentication, API keys, or multi-factor authentication. After confirming the absence of such mechanisms, the agent finalises its understanding of the authentication schema.

The final output of the agent is presented in Figure 5.1, illustrating that the agent identified tech stacks within microservices and also other software used, such as Docker.

Step	Generated next_step
1	Read the README.md file to gather detailed information about the project's purpose and scope.
2	Explore the 'services' directory to understand the different microservices implemented in the project and their roles in the hotel reservation system.
3	Examine the README or main files within key service directories such as 'recommendation', 'reservation', and 'search' to gain deeper insight into the specific functionalities and value each microservice provides within the hotel reservation system.
4	Explore the 'services' directory further by listing files in key service subdirectories like 'reservation', 'recommendation', and 'search' to identify main files or documentation that can provide more insights into their specific roles.
5	Read the content of the 'server.go' file in the 'services/reservation' directory to understand the implementation and role of the reservation service within the system.
6	Explore the 'services/recommendation' and 'services/search' directories to understand how the recommendation and search functionalities are implemented and contribute to the overall system.
7	Read the content of 'services/recommendation/server.go' to understand how the recommendation service works and contributes to the overall hotel reservation system.
8	Explore the 'services/search' directory, particularly the main service files, to understand how the search functionality is implemented and how it complements the overall hotel reservation system.
9	Review the README.md and other documentation files for any additional high-level descriptions or user-facing explanations to confirm and refine the purpose statement.

Table 5.1: Thinking steps of the Project Purpose agent for the Hotel Reservation scenario

Step	Generated next_step
1	I will start with checking the codebase for the tech stack.
2	I have identified the main programming language (Go), communication framework (gRPC), database (MongoDB), service discovery (Consul), tracing (Jaeger), and caching (Memcached) technologies from the go.mod file. Next, I plan to check for any containerization or orchestration technologies such as Docker or Kubernetes by looking into the helm-chart and kubernetes directories to confirm if they are part of the tech stack.
3	I have updated the tech stack with Helm and Kubernetes based on the presence of helm-chart and kubernetes directories. Next, I plan to check the README.md file for any additional technology mentions or confirmation of the stack components.
4	I have updated the tech stack with Docker, Docker Compose, and OpenShift based on the README.md file. The tech stack now includes Go, gRPC, MongoDB, Consul, Jaeger, Memcached, Helm, Kubernetes, Docker, Docker Compose, and OpenShift. I plan to check the services directory to see if there are any other technologies or frameworks used in the microservices implementations to further refine the tech stack.
5	The tech stack has been comprehensively identified including Go, gRPC, MongoDB, Jaeger, Consul, Memcached, Helm, Kubernetes, Docker, Docker Compose, and OpenShift. No further steps are planned as the tech stack appears complete based on the project files and description.
6	I will start with checking if the project has any authentication mechanisms.
7	I have identified that the project uses username and password authentication with password hashing using SHA-256. Next, I plan to check other service directories, especially the user service proto files or any authentication-related middleware or interceptors, to see if there are additional authentication mechanisms or details such as registration, token usage, or multi-factor authentication.
8	I have identified that the project uses username and password authentication with password hashing using SHA-256. The user service proto confirms the authentication schema with username and password fields. Next, I plan to check other service directories or middleware for any additional authentication mechanisms such as token-based authentication or multi-factor authentication to ensure completeness.
9	I will check the reservation service and other related services for any additional authentication or authorization mechanisms, such as token validation or API key usage, to ensure no other authentication methods are used in the project.
10	I have confirmed that the project uses username and password authentication with SHA-256 hashing. There is no indication of other authentication mechanisms such as OAuth, API keys, or multi-factor authentication. I will now finalize the authentication mechanisms and schema based on this information.

Table 5.2: Thinking steps of the Project Insights agent for the Hotel Reservation scenario

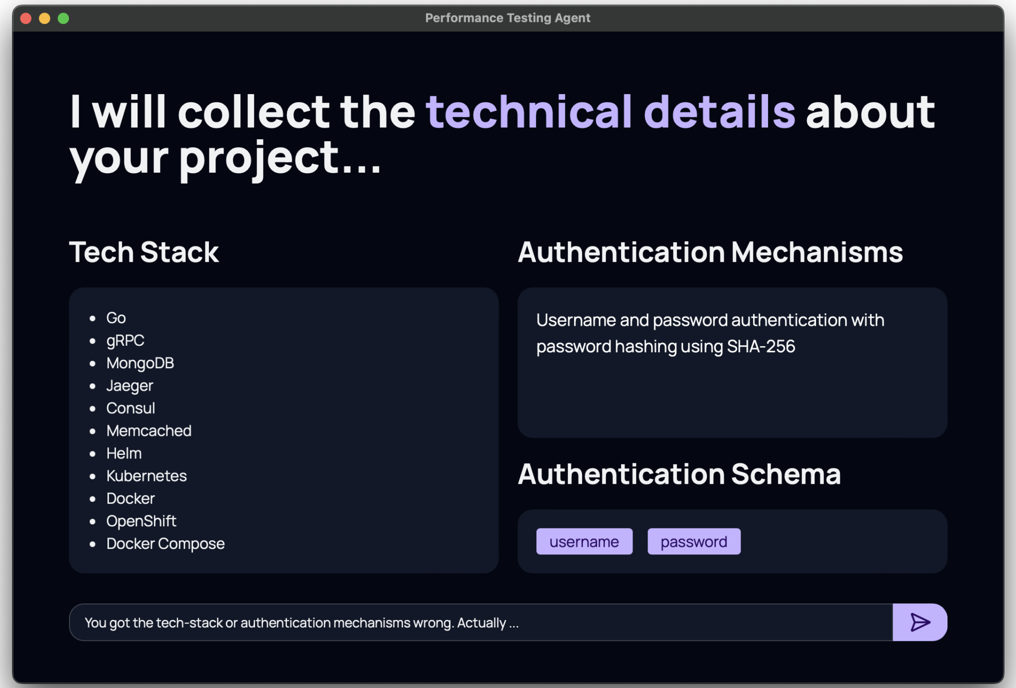

Figure 5.1: Results of the Project Insights agent for the Hotel Reservation scenario

Project Features agent execution

This agent must extract the features of the SUT, along with some metadata that provide additional insights into each extracted feature. During execution, no `next_step` messages were dumped. Two scenarios can explain this behaviour:

1. The agent did not use multiple invocations of the `Thinking` node and solely used the passed results from other agents in its initial prompt
2. The agent did not populate the `next_step` field in the state.

However, when inspecting the final output of the agent presented in Figure 5.2, it becomes evident that the agent indeed used the available MCP Server for its analysis, given the detail of information provided in the features' descriptions.

Furthermore, the agent also outputted a *Dynamic Hotel Management* feature which, based on its description, clearly results from the injected fault in the README file.

Selecting a feature

Further analysis will focus solely on the **Hotel Search** feature, which was previously identified. The performed steps from now on can be repeated for the remaining features as well, but analysing all features in detail would go beyond the scope of this experiment.

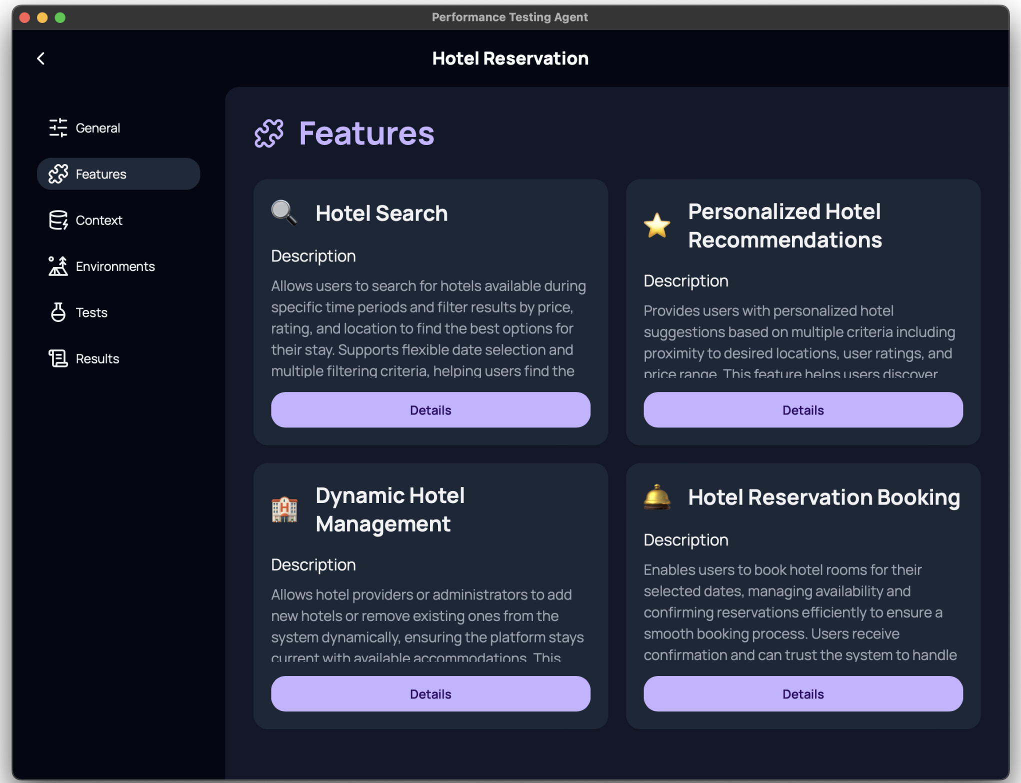

Figure 5.2: Results of the Project Features agent for the Hotel Reservation scenario

Step	Generated next_step
1	I have updated the difficulty related to Hotel Search performance under high load with detailed potential bottlenecks. Since the main performance challenge is identified and described, I plan to conclude the analysis for this feature.

Table 5.3: Thinking steps of the Feature Difficulties agent for the Hotel Reservation scenario

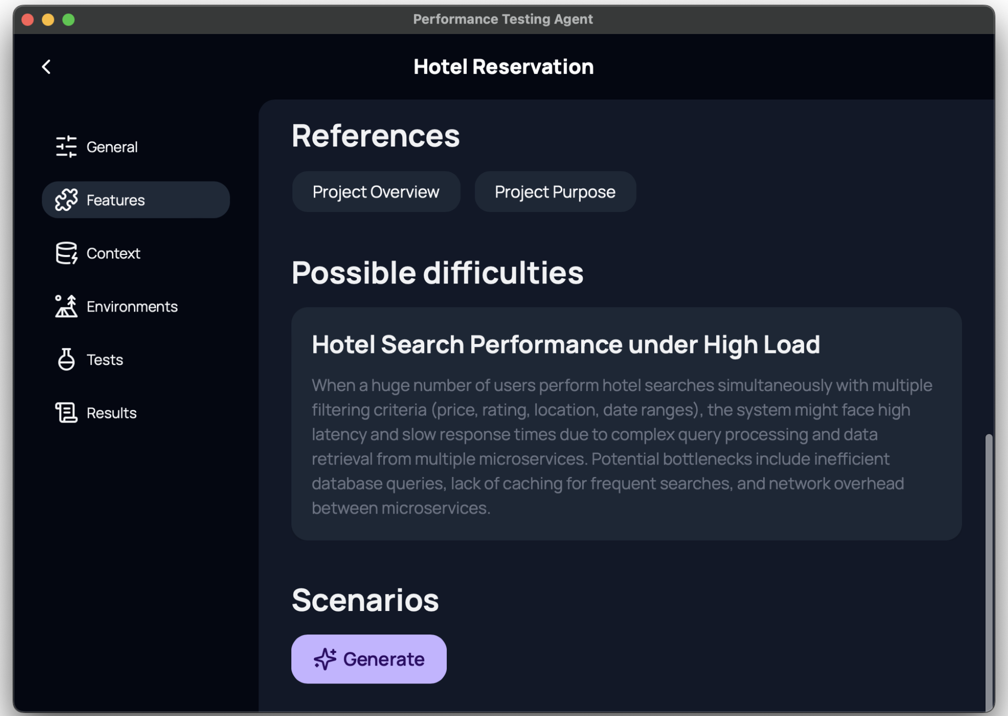

Figure 5.3: Results of the Feature Difficulties agent for the Hotel Reservation scenario

Feature Difficulties agent execution

In this step, the goal is to identify possible weaknesses within the implementation of the selected feature. The agent successfully identifies a potential bottleneck caused by the microservice architecture as presented in Figure 5.7. In terms of thinking messages, Table 5.3 highlights that the agent outputs one message, indicating that it is done with its analysis.

Feature Scenarios agent execution

The execution of this agent demonstrates its ability to generate diverse and structured fictitious user scenarios based on the results of previous agents, each reflecting different user behaviours and system interactions. The agent produced five distinct scenarios, visualised in Appendix B.1.

Collectively, these scenarios cover a range of user intents and system responses, from straightforward searches to more nuanced interactions.

Step	Generated next_step
1	Review existing scenarios for completeness and consistency, then add any missing edge cases or user interactions related to the Hotel Search feature, such as error handling for invalid date inputs or interaction with the recommendation service after search results.
2	Review all existing scenarios for completeness and consistency one last time, then finalize the scenarios for the Hotel Search feature as no further additions are planned.

Table 5.4: Thinking steps of the Feature Scenarios agent for the Hotel Reservation scenario

Table 5.4 outlines the agent's internal reasoning process. In Step 1, the agent reviews the generated scenarios for completeness and identifies potential gaps, such as missing edge cases or unhandled interactions (e.g., invalid input handling or post-search recommendations). In Step 2, after ensuring all relevant cases are addressed, the agent finalises the scenario set, indicating that no further additions are necessary. This structured approach highlights the agent's iterative refinement process and its capacity to simulate realistic user journeys that can serve as a basis for creating `Locust files`.

Project Environment agent execution

This agent created a `docker-compose.yml` file, together with additional `Dockerfiles` for the individual services, namely:

1. `services/reservation/Dockerfile`
2. `services/user/Dockerfile`
3. `services/recommendation/Dockerfile`
4. `services/search/Dockerfile`

The SUT consists of a microservice-based codebase written in Go. The agent attempted to infer the necessary build context and create one `Dockerfile` for each service individually. However, it incorrectly assumed the presence of Go module files (`go.mod` and `go.sum`), which are typically used for dependency management in Go projects. In reality, the microservices only contain a single `server.go` file with the service logic, and lacked the expected module files. As a result, the generated Docker configurations were invalid and failed to build the services correctly. This highlighted a limitation in the agent's inference mechanism and emphasised the importance of accurately detecting the structure and dependencies of the codebase before generating containerization artefacts.

The Project Environment agent itself followed a structured and iterative reasoning process to prepare the deployment environment for the system under test, as presented in Table 5.5. Initially, it identified the core services, particularly those written in Go, and began by generating `Dockerfiles` for these services, ensuring compatibility with ARM architectures, which is the architecture of the MacBook on which the agent was executed. It then constructed a `docker-compose.yml` file that integrates these services with essential dependencies such as MongoDB, Memcached, Consul, and Jaeger. As the process unfolds, the agent incrementally expands its scope to include additional services, such as user management, recommendation, and search, applying consistent patterns in Dockerfile creation and service orchestration. Throughout the process, it correctly stored key artefacts and configuration details in the provided long-term memory, including the base endpoint of the application. In later steps, the agent performs validation and refinement, reviewing environment variables, volumes, and network settings to ensure correctness. It also incorporates enhancements such as copying proto files and generating gRPC code where necessary.

However, since the generated Docker configuration did not work, the already provided Docker configuration from the `DeathStarBench` was manually inserted into the codebase to permit the remaining agents to be executed, rather than initialising the Go package manager for every microservice to make it work with the generated Docker configuration.

Selecting a goal

To enable the following agents to create a performance test that targets a specific goal based on the analysis performed so far, the tester must now select this goal. The available options are:

- Load Testing
- Stress Testing
- Scalability Testing
- Endurance Testing
- Spike Testing
- Volume Testing
- Concurrency Testing

For the following agents, *Load Testing* was selected as a goal. However, it is always possible to add more tests with different goals at a later stage.

Performance Test agent execution

In this step, the goal is to generate an executable `Locust file` that leverages the REST endpoints exposed by the SUT to carry out the performance test on them. The generated `Locust file` is shown in Appendix B.2. It becomes evident that the agent took the previously generated scenarios into account and extracted the correct endpoints from the codebase.

The agent's reasoning process for constructing a Locust performance test file, shown in Table 5.6, was methodical and goal-oriented, progressing through distinct phases of validation, enhancement, and finalisation. Initially, the agent focused on verifying the functionality of the `/hotels` endpoint by executing sample queries, ensuring the endpoint behaved as expected before integrating it into a performance testing framework. Once validated, the agent proceeded to enrich the Locust script by configuring user load parameters and defining test initiation instructions, thereby laying the groundwork for realistic load simulation. In the final step, the agent incorporated a comprehensive set of user scenarios—including

Step	Generated next_step
1	I will create Dockerfiles for the main services in the microservices architecture, focusing on Go services. Then, I will create a docker-compose.yml file that includes these services along with dependencies like MongoDB, Memcached, Consul, and Jaeger. I will ensure ARM compatibility and set the base endpoint for the project.
2	I will create a Dockerfile string for the reservation service and other main services as a single string path, then set it in long term memory. After that, I will create a docker-compose.yml file string including all services and dependencies like MongoDB, Memcached, Consul, and Jaeger, ensuring ARM compatibility. Finally, I will set the base endpoint for the project in long term memory.
3	I will create Dockerfiles for other main services such as user, recommendation, and search services following the same pattern as the reservation service. Then, I will create a comprehensive docker-compose.yml file that includes all these services along with MongoDB, Memcached, Consul, and Jaeger services. I will ensure ARM compatibility and set the base endpoint for the project in long term memory.
4	I will continue creating Dockerfiles for the recommendation and search services following the same pattern as the reservation and user services. After that, I will create a docker-compose.yml file that includes all these services plus MongoDB, Memcached, Consul, and Jaeger, ensuring ARM compatibility. Finally, I will set the base endpoint for the project in long term memory.
5	I will review the Dockerfiles and docker-compose.yml file to ensure all necessary services are included and properly configured. I will also verify that environment variables, volumes, and network settings are correctly set for the microservices and dependencies. If everything is correct, I will finalize the setup and mark the task as done.
6	I will update the Dockerfiles to include copying proto files and generating gRPC code if necessary, and ensure environment variables for MongoDB, Memcached, and Consul are passed in the docker-compose.yml. I will also verify volumes and network settings. Then, I will finalize the Dockerfiles and docker-compose.yml for the microservices and dependencies.

Table 5.5: Thinking steps of the Project Environment agent for the Hotel Reservation scenario

Step	Generated next_step
1	I will test the /hotels endpoint with sample query parameters to verify it works as expected before finalizing the locust performance test script.
2	I will enhance the locust performance test by adding user load configuration and test start instructions. Then I will finalize and save the locust file for the hotel search feature load testing.
3	I have updated the locust file with detailed user scenarios for hotel search under load, including filters, no filters, no availability, and flexible dates. The locust file is ready for execution. The next step is to consider the test execution and monitoring phase to validate performance under load.

Table 5.6: Thinking steps of the Performance Test agent for the Hotel Reservation scenario

searches with filters, without filters, with no availability, and with flexible dates—into the Locust file. This ensured that the performance test would reflect diverse usage patterns as generated by the *Feature Scenarios* agent.

PPTAM Wrapper agent execution

Figure 5.4 shows the generated PPTAM test steps required for executing the performance test. It reveals that the agent successfully took into account the goal of the performance test, namely *Load Testing*, and constructed the test steps accordingly.

Regarding the thinking process shown in Table 5.7, the agent first enhances the performance test by introducing additional load steps with gradually increasing user loads. This decision demonstrates an understanding of the primary objective of load testing, which is to evaluate system behaviour under stress and identify potential performance bottlenecks.

In the second step, the agent retrieves the current summary of the PPTAM test plan to ensure that the planned test scenarios align with the intended testing goals. This reflective step enables the agent to verify the completeness of the test plan before proceeding.

Finally, in the third step, the agent concludes the planning phase, recognising that the test plan now includes sufficient load variations to effectively assess the hotel search feature's performance under different traffic conditions. This sequence highlights the agent's ability to refine and validate its strategy iteratively, ensuring alignment with the overall testing objective.

The generated PPTAM configuration seen in Figure 5.4 accurately describes the goal of a Load Test.

Setting the operational profile

The tool enables performance testers to create an operational profile of the SUT in plain text, highlighting special cases and the average load on the system. For this test, the following sample operational profile was provided:

I have on average around 300 daily users, however, on some days that number can climb up to 1000.

Note that the operational profile is not important for the execution of the last agent. Moreover, it is necessary to determine how the agent embeds the operational profile within the report and how it links the raw test results to that operational profile.

Step	Generated next_step
1	Add additional load steps with increasing user load to simulate higher traffic and observe system behavior under stress. This will help identify performance bottlenecks and system limits for the hotel search feature.
2	Retrieve and provide the current PPTAM test plan summary to review the planned load tests for the hotel search feature before concluding the performance test planning.
3	The current PPTAM test plan includes three load steps with increasing user loads to simulate different traffic levels for the hotel search feature. I will now conclude the performance test planning as the test plan sufficiently covers load testing scenarios to evaluate response times and system behavior under varying user loads.

Table 5.7: Thinking steps of the PPTAM Wrapper agent for the Hotel Reservation scenario

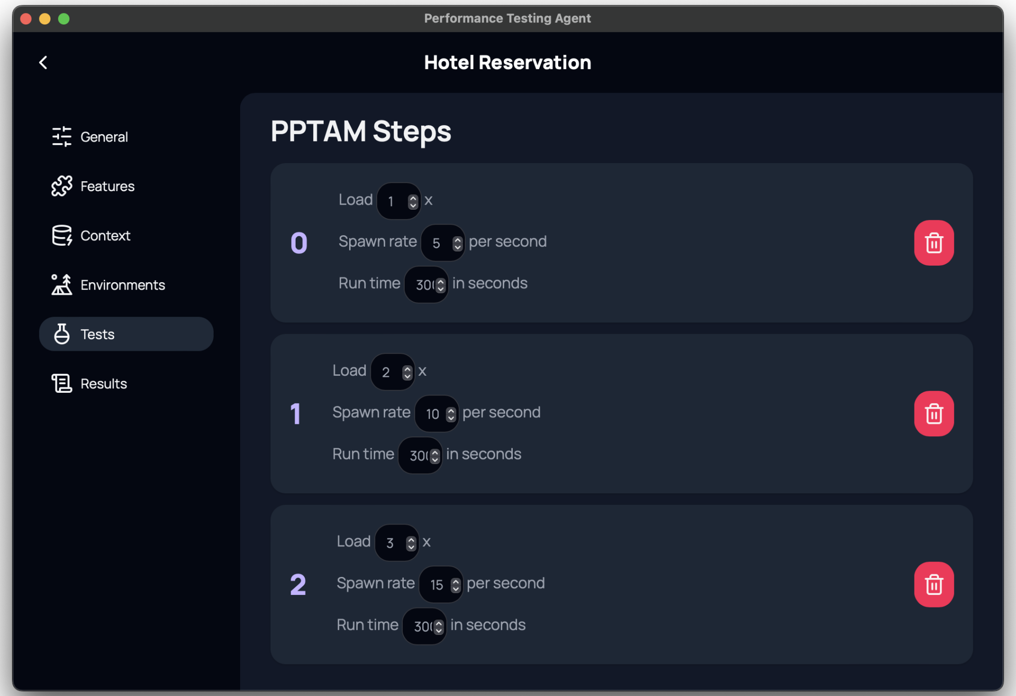

Figure 5.4: Results of the PPTAM Wrapper agent for the Hotel Reservation scenario

Step	Generated next_step
1	I will start with planning the structure of the report and gathering the necessary information from the knowledge shelf.
2	Review the report sections for clarity, completeness, and alignment with the project goals and performance test objectives. Ensure that all findings are clearly linked to the operational profile and that recommendations are actionable. If any gaps or ambiguities are found, update the relevant sections for improved precision and readability. If the report is already comprehensive and clear, finalize it and mark the process as complete.
3	The report sections are clear, comprehensive, and directly address the operational profile, test results, and performance requirements for the Hotel Search feature. All findings and recommendations are well-aligned with the project goals and performance test objectives. As the report is already complete and no further gaps or ambiguities are identified, the next step is to finalize the report and mark the process as done.

Table 5.8: Thinking steps of the Result Interpreter agent for the Hotel Reservation scenario

Result Interpreter agent execution

The agent was tasked with interpreting the raw performing test results and putting them into perspective with the predefined operational profile. As shown in Table 5.8, its operation begins by ingesting the raw data from performance tests and aligning it with the expected usage patterns defined in the operational profile. The agent then analyses key performance indicators—such as response times, throughput, and error rates—across various test scenarios. It puts these metrics into context by comparing them against the operational expectations, identifying whether the system meets, exceeds, or falls short of the required performance levels. Finally, it summarises the findings into a report, complete with interpretations, comparisons, and actionable recommendations.

The final report, presented in Appendix B.3, illustrates strong technical clarity and methodological integrity, organised around the operational profile, which is essential for meaningful interpretation of performance metrics. Key indicators, including the median, 90th percentile, and maximum response times, are present. Moreover, the report provides in-depth insights beyond raw data and recommends actions for the performance tester, such as conducting regular performance evaluations and stress testing at higher loads.

5.4.2 Social Network

The outcomes of all agent executions will be presented and analysed regarding their handling of the injected faults and the SUT as a whole, similar to the Hotel Reservation scenario. Furthermore, similarities and differences between the agent executions of both SUTs will be examined.

Project Purpose agent execution

In executing the Project Purpose agent on the *Social Network* application, the agent follows a comprehensible reasoning process to derive a description of the SUT, as outlined in Table 5.9. It begins by analysing the README.md file to establish a foundational understanding of the project's goals and scope. It then explores the src directory to identify the core microservices and their roles within the system. A key step in this process is examining the social_network.thrift file, which defines the service interfaces and interactions between components. This file offers crucial insights into the system's architecture and how its services interact. Based on this exploration, the agent synthesises a high-level description that captures the system's purpose, emphasising its microservices-based architecture, support for user interactions, content sharing, and scalability for large-scale social networking scenarios.

The following result was produced:

> This project is a microservices-based social network platform designed to facilitate user registration, social connections through unidirectional follow relationships, and content sharing via posts. It provides a realistic social networking experience where users can compose posts, mention other users, share media, and view personalized timelines including home and user timelines. The architecture emphasizes scalability, modularity, and efficient inter-service communication using RPC, making it suitable for handling large social graphs and high request volumes. The system is valuable for enabling social interaction and content dissemination in a distributed environment and serves as a benchmark or research tool for microservices-based social network implementations.

When comparing this execution to the one performed on the *Hotel Reservation* application, several similarities emerge. In both cases, the agent begins with high-level documentation to establish context, then drills down into the codebase to examine service-specific files or definitions. The agent's reasoning is iterative and hierarchical, moving from general to specific, and it concludes by refining its understanding through revisiting documentation or interface definitions.

However, there are also notable differences. The hotel reservation scenario involves a deeper traversal of the codebase, including multiple service directories and specific implementation files, such as server.go, which reflects a more granular inspection of service logic. In contrast, the social network scenario relies more heavily on the `.thrift` interface definitions and a smaller number of reasoning steps, suggesting that the architectural and functional insights were more centralised and accessible through fewer files. This difference may stem from the way the two applications are structured: the hotel reservation system distributes logic across multiple services with distinct responsibilities. In contrast, the social network application encapsulates much of its service interaction logic in the .thrift definitions.

Overall, both executions demonstrate the agent's ability to adapt its exploration strategy based on the structure and documentation of the target system.

Project Insights agent execution

The agent's thinking steps, presented in Table 5.10, display a consistent and logical progression. In both scenarios, the agent begins by examining core directories and configuration files that are likely to reveal the primary languages, frameworks, and supporting technologies. For the hotel reservation application, the agent begins with the go.mod file and quickly identifies the essential backend technologies. Then, it checks for orchestration and containerization tools by examining specific directories. The process is similar in the social network application: the agent first explores the frontend directory, then inspects deployment-

Step	Generated next_step
1	Read the README.md file to gather detailed information about the project's goals, scope, and intended user value.
2	Explore the src directory to identify key components and services implemented in the project to better understand the architecture and user value.
3	Review the README.md and explore the social_network.thrift file to understand the service definitions and interactions between microservices, which will clarify the system's capabilities and user value.
4	Summarize the purpose based on the detailed service definitions and README, and finalize the project purpose description as the information is comprehensive.

Table 5.9: Thinking steps of the Project Purpose agent for the Social Network scenario

related directories, and finally delves into configuration files to uncover further details about the tech stack.

After compiling a comprehensive list of technologies, the agent transitions to analysing authentication mechanisms as defined by its architecture. It initiates this stage by searching for evidence of login or credential-management functionality, such as service proto files or configuration schemas. In both workflows, the agent methodically checks additional service directories and middleware to ensure no authentication or authorisation method is overlooked.

The results of this agent execution, shown in Figure 5.5, were double-checked with the codebase. All generated entries are valid for both the tech stack and the authentication mechanisms.

Project Features agent execution

Similar to the execution of this agent for the hotel reservation project, no `next_step` messages were logged in this case either.

However, a thorough investigation of the codebase must have been executed, as the agent would otherwise not be able to describe the identified features presented in Figure 5.6.

Since all technical details were removed from the README as part of the fault-injection step, the agent's output can now be assessed in terms of its quality using the original README file as a comparison. In the original README file, the following features were listed:

- Create text post (optional media: image, video, shortened URL, user tag)
- Read post
- Read entire user timeline
- Receive recommendations on which users to follow
- Search database for user or post
- Register/Login using user credentials
- Follow/Unfollow user

Step	Generated next_step
1	I will start with checking the codebase for the tech stack.
2	Explore the media-frontend directory to identify frontend technologies used in the project.
3	Review the helm-chart directory to check for Kubernetes or Helm usage for deployment orchestration.
4	Check the config directory for configuration files that might indicate use of other technologies or frameworks such as databases, message brokers, or monitoring tools.
5	Add Jaeger to the tech stack for distributed tracing and monitoring based on the presence of jaeger-config.yml in the config directory. Then, check the content of service-config.json for any other technologies or services used.
6	The tech stack has been updated with Docker, Docker Compose, Python 3.5+, Lua, Nginx, Thrift RPC, C++, Redis, MongoDB, Memcached, OpenResty, mcrouter, Helm, and Jaeger. Next, I plan to check the helm-chart/socialnetwork directory for Helm chart files to confirm Kubernetes usage and possibly identify other technologies or configurations.
7	I will start with checking if the project has any authentication mechanisms.
8	The authentication mechanisms and schema have been updated to reflect username and password authentication with salted and hashed passwords stored in MongoDB, JWT token generation upon login, and token-based authentication using an X-Token header for subsequent requests. No further steps planned as the authentication mechanisms appear fully described.

Table 5.10: Thinking steps of the Project Insights agent for the Social Network scenario

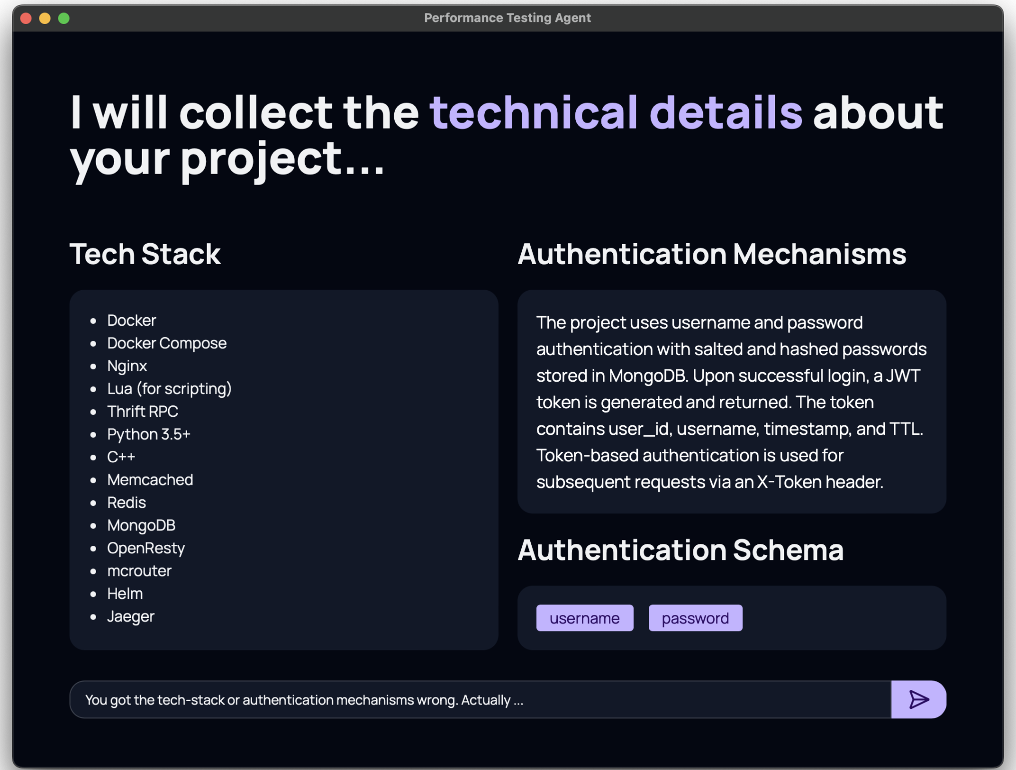

Figure 5.5: Results of the Project Insights agent for the Social Network scenario

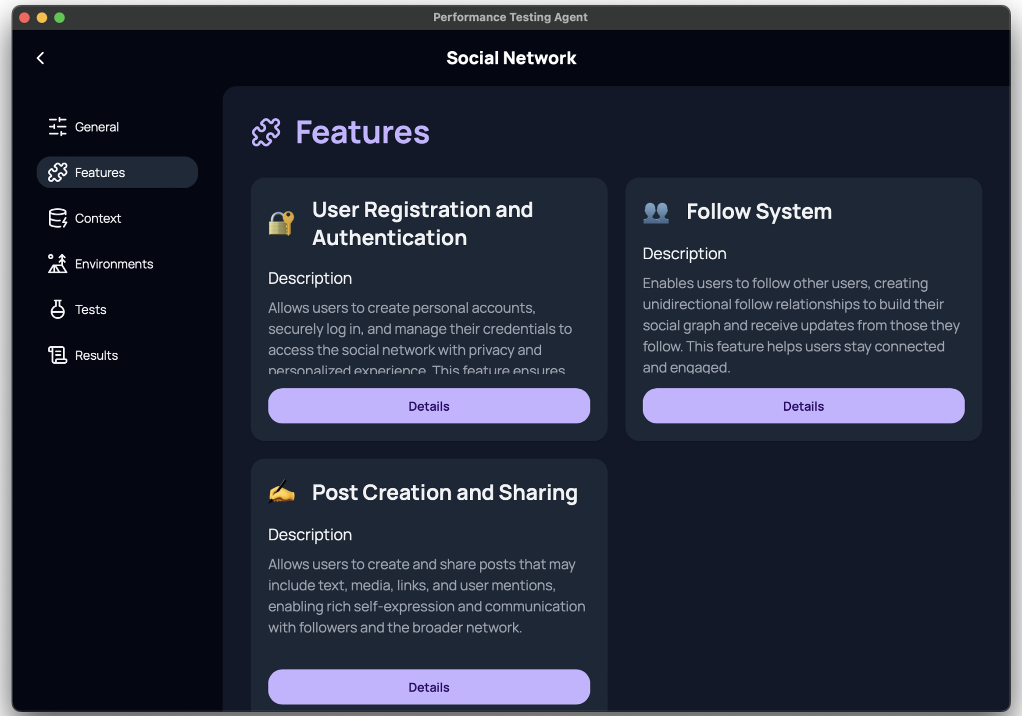

Figure 5.6: Results of the Project Features agent for the Social Network scenario

Taking into account the agent's results, the following features were not identified at all:

- Read entire user timeline
- Receive recommendations on which users to follow
- Search database for user or post

Note that the *Read post* feature is indirectly explained in the description of the *Follow System* feature, stating it allows to receive updates from those they follow.

Selecting a feature

Similar to the hotel reservation scenario, a specific feature for detailed analysis must be selected. For the sake of the remaining agent executions, the **Post Creation and Sharing** has been chosen.

Feature Difficulties agent execution

For this agent execution, the information revealed by the **next_step** messages, as depicted in Table 5.11, is sparse, similar to the execution with the hotel reservation project.

However, one possible difficulty was identified as shown in Figure 5.7. It addresses the microservice architecture, which may serve as a bottleneck when creating a post or utilising the sharing feature.

Step	Generated next_step
1	I have added the main difficulty related to scalability and performance issues in the post creation and sharing feature due to multiple RPC calls and media handling under high load. Since this covers the critical aspects of the feature's performance challenges, I plan no further steps.

Table 5.11: Thinking steps of the Feature Difficulties agent for the Social Network scenario

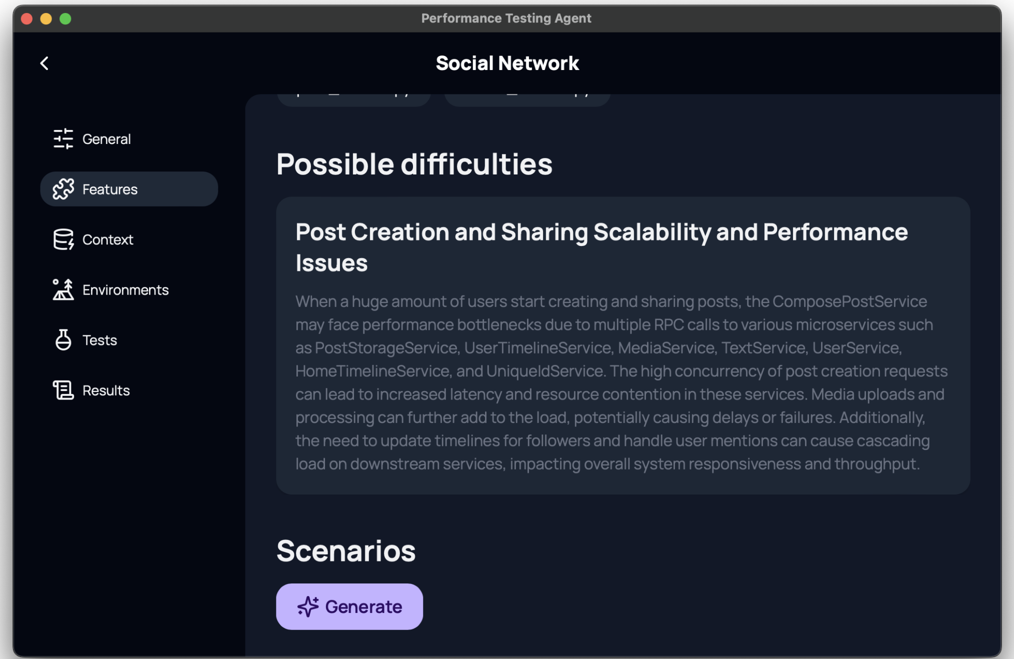

Figure 5.7: Results of the Feature Difficulties agent for the Social Network scenario

Step	Generated next_step
1	Review existing scenarios for completeness and consistency, then finalize the scenario set for the 'Post Creation and Sharing' feature.

Table 5.12: Thinking steps of the Feature Scenarios agent for the Social Network scenario

Step	Generated next_step
1	I will create a Dockerfile as a string (not a list) and set it in the long term memory. Then I will create a docker-compose.yml file that includes services for the social network app, MongoDB, Redis, Memcached, Jaeger, and Nginx, based on the project description and README instructions. I will also set the base endpoint to http://localhost:8080 as indicated in the README.
2	I have created a Dockerfile that installs necessary dependencies and runs the social network app, and a docker-compose.yml file that defines services for the app, MongoDB, Redis, Memcached, Jaeger, and Nginx with proper port mappings and volumes. The base endpoint is set to http://localhost:8080. The next step is to consider if any environment variables or additional configuration files need to be added to the docker-compose or Dockerfile, but based on the README and config files, the current setup should be sufficient to start the project in a Docker environment on Apple Silicon. Therefore, I consider this task done.

Table 5.13: Thinking steps of the Project Environment agent for the Social Network scenario

Feature Scenarios agent execution

The result of this agent can be seen in Appendix B.4. Similar to the execution with the hotel reservation SUT, this agent also suggests scenarios that test edge cases to uncover hidden performance issues while ensuring that the entire functionality of the feature under test is represented within the generated scenarios, as shown in Table 5.12. Overall, the execution of the agent in this project is highly similar to that of the hotel reservation feature.

Project Environment agent execution

In contrast to the `docker-compose.yml` and `Dockerfile` created for the first scenario, the Docker configuration generated for this scenario was at least syntactically valid. However, the service `nginx` was not configured at all, meaning no configuration file was provided, hence it did not start. Although the agent considered adding additional configuration, as shown in Table 5.13, it ultimately decided against it after reviewing the README and other configuration files. As a result, the leading service of this project, `social network app`, also did not start as it depends on `nginx`. Therefore, this agent has failed its task of containerising the SUT.

Similar to the first scenario, the original Docker files were restored in the codebase to enable testing the remaining agents.

Step	Generated next_step
1	I will verify the correctness of the login and post creation endpoints by calling them with test data to ensure they work as expected before finalizing the locust performance test script.
2	I will finalize the locust file with the current implementation for load testing the post creation feature, including user login and different post creation scenarios. Then I will mark the task as done.

Table 5.14: Thinking steps of the Performance Test agent for the Social Network scenario

Step	Generated next_step
1	Add a higher load test step to simulate increased user traffic for post creation and sharing to observe system behavior under stress.
2	The PPTAM test plan for load testing post creation and sharing feature is complete with three load levels. No further steps are planned.

Table 5.15: Thinking steps of the PPTAM Wrapper agent for the Social Network scenario

Selecting a goal

As the fundamental analysis phase of the SUT is now complete, the performance tester must determine a goal for the performance test, as already seen during the execution of the hotel reservation scenario. To keep the factors the same for both scenarios, which increases comparability between them, **Load Testing** was again chosen as the performance test goal.

Performance Test agent execution

The generated Locust file can be found in Appendix B.5. It can be observed that the agent successfully understood the authentication mechanisms of this project by logging in as the predefined test user first. The generated steps within the Locust file mimic the previously generated scenarios, likewise observed when executing this agent for scenario 1. The agent performed exactly as intended.

PPTAM Wrapper agent execution

Table 5.15 shows how the agent constructed the PPTAM configuration, revealing it chose a step-by-step approach to append one step at a time to the final configuration. The result presented in Figure 5.8 reveals a very similar configuration to the one created for scenario 1. However, this configuration aims to increase the load on the SUT much quicker.

Setting the operational profile

The operational profile was copied from the Hotel Reservation scenario to limit varying factors between both scenario executions. Namely: *I have on average around 300 daily users, however, on some days that number can climb up to 1000.*

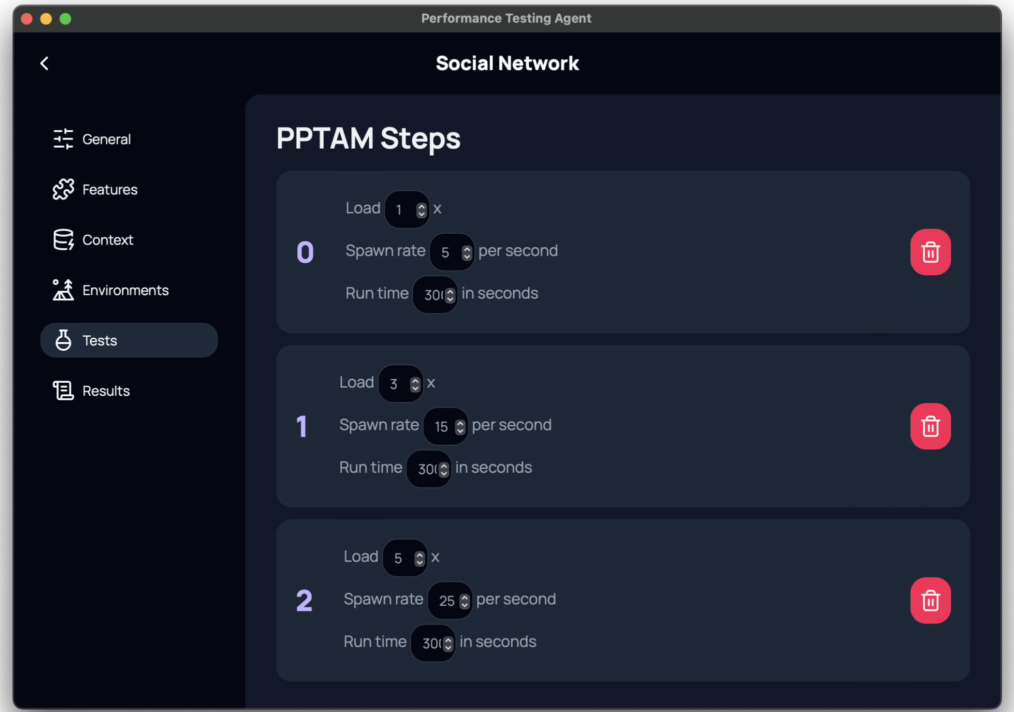

Figure 5.8: Results of the PPTAM Wrapper agent for the Social Network scenario

Step	Generated next_step
1	I will start with planning the structure of the report and gathering the necessary information from the knowledge shelf.
2	Review the report for clarity, completeness, and alignment with the project goals and performance test objectives. Ensure that all findings are clearly linked to the operational profile and that recommendations are actionable. If any gaps or ambiguities are found, update the relevant sections for improved precision and completeness.
3	The report has been reviewed for logical flow, completeness, and clarity. All sections are well-aligned, and the findings and recommendations are clearly linked to the operational profile and test results. No further refinements are necessary unless new data or requirements emerge. The next step is to mark the report as complete.

Table 5.16: Thinking steps of the PPTAM Wrapper agent for the Social Network scenario

Result Interpreter agent execution

The final report for this scenario can be found in Appendix B.6. It follows the same structure as outlined in scenario 1, providing readers with a clear and comprehensible overview of the performance test, while ensuring the text remains concise and easy to read. Additionally, recommendations are provided, similar to those in the report from scenario 1. Furthermore, the thinking steps presented in Table 5.16 largely overlap with those from the previous scenario.

Chapter 6

Discussion

This chapter discusses the results obtained in the evaluation and the threats of validity.

6.1 Evaluation interpretation

This section provides a critical analysis of the evaluation results obtained from executing the developed tool on two real-world microservice-based applications. The goal is to assess how effectively the agentic AI system performed in generating, implementing, and interpreting performance tests. Each scenario is examined individually to highlight the strengths and limitations of the tool in different contexts, followed by a synthesis of the overall findings.

6.1.1 Interpretation of the Hotel Reservation scenario

The evaluation of the *Hotel Reservation* scenario demonstrated the tool's strong capability to autonomously generate and execute a complete performance test with minimal human intervention. The agents successfully navigated the codebase, extracted relevant features, and constructed realistic user scenarios that reflected diverse usage patterns. Notably, the *Feature Scenarios* agent produced a well-rounded set of test cases, including edge cases such as searches with no availability or flexible dates, which contributed to a comprehensive performance evaluation.

Despite the deliberate injection of misleading information, such as a fabricated feature in the README and a decoy service folder, the agents showed a mixed ability to validate and cross-reference data. For instance, the *Project Purpose* agent incorporated the non-existent chat functionality into its system description, indicating a reliance on documentation without sufficient verification against the actual codebase. This highlights a current limitation in the agents' reasoning depth, particularly in distinguishing between declared and implemented features.

The Dockerization process, handled by the *Project Environment* agent, revealed another area for improvement. The agent assumed the presence of Go module files, which were absent, resulting in invalid Docker configurations. This misstep underscores the need for more robust detection mechanisms when inferring build requirements from unconventional or minimalistic code structures.

On the positive side, the *Performance Test* agent generated a valid and executable Locust file that accurately reflected the previously defined user scenarios. The test execution, facilitated by the *PPTAM Wrappe*r agent, was well-aligned with the selected goal of load testing. The *Result Interpreter* agent then produced a detailed and insightful report, effectively contextualising the raw performance metrics against the operational profile. The report confirmed that the system exceeded performance expectations, with low response times, zero failures, and throughput well above the required threshold.

Overall, the *Hotel Reservation* scenario validated the tool's end-to-end functionality and highlighted its potential for real-world application. While the agents demonstrated impressive autonomy and contextual understanding, the evaluation also revealed areas where deeper validation and improved inference could enhance reliability and robustness.

6.1.2 Interpretation of the Social Network scenario

The evaluation of the Social Network scenario further validated the tool's ability to autonomously generate and execute performance tests in a complex, microservice-based environment. Compared to the Hotel Reservation scenario, this application presented a broader and more interconnected service architecture, offering a valuable opportunity to assess the tool's scalability and adaptability.

The agents demonstrated a high degree of autonomy and contextual awareness, particularly in navigating the codebase and identifying relevant features. Despite the deliberate removal of key documentation, as part of the fault injection, the agents were still able to extract meaningful insights from the source code alone. This was evident in the *Project Purpose* and *Project Insights* agents, which successfully reconstructed the system's purpose and technical stack, including nuanced authentication mechanisms such as JWT-based token handling.

However, the `Project Features` agent showed some limitations in completeness. While it correctly identified core features like post creation and user registration, it failed to detect others, such as timeline viewing and user recommendations features that were originally documented but removed for the evaluation. This suggests that while the agent can infer functionality from implementation details, it still benefits significantly from supplementary documentation when available.

The *Project Environment* agent again struggled with Dockerization, similar to the first scenario. Although the generated configuration was syntactically valid, it lacked critical components, mainly the Nginx configuration, resulting in a non-functional deployment.

On the other hand, the `Performance Test` agent performed exceptionally well. It correctly incorporated authentication flows and generated a Locust file that reflected realistic user interactions, including media uploads and user mentions. The `PPTAM Wrapper` agent constructed a suitable load test plan, and the *Result Interpreter* agent produced a comprehensive report that presented the results against the operational profile.

In comparison to the Hotel Reservation scenario, the Social Network scenario highlighted the tool's ability to scale to more complex systems while maintaining effectiveness. However, it also exposed a greater reliance on documentation for feature discovery and a continued need for improvement in environment setup automation.

6.1.3 Overall tool effectiveness

The evaluation of the developed tool across two distinct microservice-based applications, namely Hotel Reservation and Social Network, demonstrated its strong potential for automating performance testing through agentic AI. In both scenarios, the tool was able to autonomously generate, execute, and interpret performance tests with minimal human intervention, validating its core design goals. To summarise both scenarios, a list of strengths and weaknesses was created:

Strengths:

1. *End-to-End Automation*: The tool successfully automated the entire performance testing lifecycle, from scenario generation to result interpretation, without requiring manual scripting or configuration.

2. *Contextual Understanding*: Even when limited to static code analysis via solely leveraging the *Codebase Explorer MCP Server*, the agents were able to extract meaningful insights about the system under test, including architecture, features, and authentication mechanisms.

3. *Robust Scenario Generation*: The *Feature Scenarios* agent consistently produced diverse and realistic user flows, including edge cases, which enhanced the quality of the generated performance tests.

4. *Accurate Test Execution and Analysis*: The *Performance Test* and *PPTAM Wrapper* agents generated valid Locust files and test plans aligned with the selected testing goals. The *Result Interpreter* agent provided clear, actionable reports that put the raw metrics into perspective in relation to the predefined operational profiles.

Weaknesses:

1. *Validation of Documentation*: In both scenarios, agents occasionally relied too heavily on documentation without verifying its accuracy against the actual codebase. This led to the inclusion of non-existent features in the system description. This is a common problem in the industry, as not every company continuously maintains their documentation of their projects.

2. *Dockerization Challenges*: The Project Environment agent struggled to generate valid Docker configurations, particularly when expected files, e.g., go.mod, were missing or when services required additional setup, e.g., Nginx.

3. *Feature Detection Completeness*: While the agents identified core features effectively, some secondary or less explicitly defined features were missed, especially when documentation was sparse or removed.

6.2 Threats to Validity

To ensure a critical and transparent evaluation of the presented work, this section discusses potential threats to the validity of the results. These are categorised into four commonly accepted dimensions: internal, external, construct, and conclusion validity.

1. **Internal validity**: Internal validity focuses on whether the observed results can be reliably linked to the developed tool and its agentic AI architecture, rather than being influenced by external or uncontrolled factors. In this thesis, various steps were taken to isolate the tool's effects, including conducting experiments in a controlled environment. Nevertheless, some uncontrolled variables might still have impacted the findings:

- **LLM non-determinism**: The underlying language models GPT-4.1 and GPT-4.1-mini show stochastic behaviour, meaning that repeated executions of the same agent may yield slightly different outputs. This introduces variability that is difficult to fully control.

2. **External Validity** External validity refers to the extent to which the study's findings are applicable to contexts beyond the specific test cases examined. Although the evaluation focused on two real-world, microservice-based applications from the DeathStarBench suite, the results may not be representative of all kinds of software systems.

   - **Domain specificity**: The tool was tested only on web-based, microservice architectures. Its effectiveness on monolithic systems was not evaluated within this thesis.
   - **Tool configuration**: The evaluation relied solely on the *Codebase Explorer MCP Server*. In real-world scenarios, additional context sources (e.g., Jira, GitHub, Swagger) may be available and will most likely significantly influence the tool's performance and output quality.

3. **Construct Validity** Construct validity examines whether the evaluation accurately measures its intended objectives. In this thesis, the focus was on assessing the feasibility and effectiveness of using agentic AI for automating performance testing. Although the assessment concentrated on generated artefacts, there are still some limitations to consider:

   - **Proxy metrics**: The quality of generated Locust files and Docker configurations was used as a proxy for agent effectiveness. However, these artefacts may not fully capture the agents' reasoning capabilities or robustness.
   - **Limited user feedback**: The evaluation did not include usability studies or feedback from real developers, which could have provided additional insights into the practical utility of the tool.

4. **Conclusion Validity** Conclusion validity refers to the robustness of the inferences made from the data. Although the findings indicate that the tool can independently create and perform performance tests, various factors may influence the dependability of these conclusions. Some exmples are:

   - **Sample size**: Only two systems were evaluated, which limits the statistical power of the findings.
   - **Single-run evaluations**: Each agent was executed once per scenario. Repeated runs could have revealed variability in outputs or uncovered edge cases.

Despite these limitations, the evaluation provides a strong initial indication of the tool's capabilities and highlights promising directions for future refinement and broader validation.

Chapter 7

Conclusion and Further Studies

This thesis explored the feasibility and effectiveness of using agentic AI to automate the end-to-end process of software performance testing. By leveraging LLMs, LangGraph agents, and a modular architecture built around MCP servers and the PPTAM framework, a comprehensive desktop application was developed to autonomously generate, execute, and interpret performance tests with minimal human intervention.

The tool was evaluated on two real-world microservice-based applications from the DeathStarBench suite. The results demonstrated that the system could successfully extract meaningful insights from the codebase, generate realistic performance test scenarios, execute them using a dynamic test runner, and produce actionable reports. Even when relying solely on static code analysis, the agents demonstrated strong contextual understanding and autonomy. The modular architecture, based on LangGraph and MCP servers, proved to be adaptable to different system architectures.

However, the evaluation also revealed several limitations. The *Project Environment* agent struggled with Dockerization in the absence of standard configuration files, and some agents occasionally relied too heavily on documentation without verifying its accuracy against the codebase. These findings highlight areas for future improvement.

Suggestions for Further Studies

To build upon the foundation laid by this thesis, the following directions are recommended for future research and development:

1. **Enhance the Project Environment Agent with Execution Tools**
   Currently, the *Project Environment* agent generates Dockerfiles based on static analysis. By integrating execution tools, the agent could test the Dockerfiles it creates and inspect console output during container startup. This feedback loop would allow the agent to refine its configurations, significantly improving accuracy and robustness iteratively.

2. **Introduce a Dedicated Orchestration Agent**
   The current system uses a Tauri-based desktop application to orchestrate agent execution. A more scalable and modular approach would be to implement a tenth agent responsible for orchestrating the others using agent-to-agent communication principles. This would enable fully autonomous workflows and lay a foundation for headless or even server-based deployments.

3. **Expand Contextual Understanding with Additional MCP Servers**
   While the tool already supports integration with GitHub, Jira, and Swagger via MCP servers, future work could explore additional sources such as CI/CD pipelines, runtime logs, or telemetry data. This would allow agents to reason not only about the static structure of the system but also about its dynamic behavior in production.

4. **Improve Feature Validation and Redundancy Checks**
   Agents should be enhanced with mechanisms to cross-validate information from multiple sources. For example, if a feature is mentioned in documentation but not implemented in the codebase, the system should flag it as potentially outdated or incorrect.

5. **User-Centric Evaluation and Usability Studies**
   Future work should include usability testing with real developers to assess the tool's practical utility, user experience, and learning curve. Feedback from practitioners could guide improvements in the UI, agent transparency, and customisation options.

6. **Integration with CI/CD Pipelines**
   To support continuous performance testing, the tool could be extended to run in CI/CD environments. This would allow for automated regression testing and performance monitoring with each code change.

These directions aim to further enhance the autonomy, reliability, and applicability of agentic AI in software performance testing, thus moving towards more intelligent and adaptive testing systems in the future.

Bibliography

[1] Viktoriia Mykytas. The role of artificial intelligence in economic transformation: From automation to the data economy. *Three Seas Economic Journal*, 6:66–73, 06 2025.

[2] Chengqing Zong Yang Zhao, Jiajun Zhang. Transformer: A general framework from machine translation to others | machine intelligence research, 6 2023. [Online; accessed 2025-07-09].

[3] Lawrence Fisher. How ai/llms can help, hinder developers – communications of the acm, 6 2025. [Online; accessed 2025-07-09].

[4] Glenn Zorpette. Large language model performance raises stakes - ieee spectrum, 7 2025. [Online; accessed 2025-07-09].

[5] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. Performance engineering for microservices: Research challenges and directions. 04 2017.

[6] Tingshuo Miao, Asif Imtiaz Shaafi, and Eunjee Song. Systematic mapping study of test generation for microservices: Approaches, challenges, and impact on system quality. *Electronics*, 14(7), 2025.

[7] Nikitha Medeshetty, Ahmad Nauman Ghazi, Sadi Alawadi, and Fahed Alkhabbas. From requirements to test cases: An nlp-based approach for high-performance ecu test case automation, 05 2025.

[8] Waleed Abdeen, Xingru Chen, and Michael Unterkalmsteiner. An approach for performance requirements verification and test environments generation. *Requirements Engineering*, 28(1):117–144, 2023.

[9] Shravan Pargaonkar. A comprehensive review of performance testing methodologies and best practices: Software quality engineering. *International Journal of Science and Research (IJSR)*, 12:2008–2014, 11 2023.

[10] Soren Lauesen. *Software Requirements: Styles and Techniques*. Addison Wesley, 2002.

[11] Mike Cohn. Advantages of the "as a user, i want" user story template., April 2008, accessed October 31, 2020. `https://www.mountaingoatsoftware.com/blog/advantages-of-the-as-a-user-i-want-user-story-template`.

[12] Scott O. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, March 1997.

[13] Alberto Avritzer, Ricardo Britto, Catia Trubiani, Matteo Camilli, Andrea Janes, Barbara Russo, André van Hoorn, Robert Heinrich, Martina Rapp, Jörg Henß, and Ram Kishan Chalawadi. Scalability testing automation using multivariate characterization and detection of software performance antipatterns. *J. Syst. Softw.*, 193:111446, 2022.

[14] Rizwan Bahrawar Khan. Comparative study of performance testing tools: apache jmeter and hp loadrunner, 2016.

[15] Anna Trudova, Michal Dolezel, and Alena Buchalcevova. Artificial intelligence in software test automation: A systematic literature review. pages 181–192, 01 2020.

[16] Dhaya Sindhu Battina. Artificial intelligence in software test automation: A systematic literature review. *SSRN Electronic Journal*, 6:1329–1332, 12 2019.

[17] Mahshid Helali Moghadam. Machine learning-assisted performance testing. 08 2019.

[18] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, 17, 06 2008.

[19] Introduction - model context protocol. [Online; accessed 2025-07-07].

[20] Jorge Contreras. Understanding mcp servers: The model context protocol explained - dev community, 5 2025. [Online; accessed 2025-07-07].

Appendix A

Search query for the mapping study

```
TITLE-ABS-KEY ( "performance testing" OR "load testing" OR "stress testing"
OR "scalability testing" OR "throughput testing" OR "latency testing" )
AND
TITLE-ABS-KEY ( "artificial intelligence" OR "AI" OR "machine learning"
OR "deep learning" OR "agent*" OR "LLM" OR "large language model*"
OR "language model*" )
AND
TITLE-ABS-KEY ( "software" OR "software system*" OR "software engineering"
OR "application*" OR "microservice*" OR "web service*" )
AND
PUBYEAR > 2018
AND
( LIMIT-TO ( DOCTYPE, "cp" ) OR LIMIT-TO ( DOCTYPE, "ar" ) )
```

Appendix B

Textual results returned from agent executions

B.1 Generated scenarios for the Hotel Reservation scenario

- Alice (Basic Hotel Search with Date and Filters)
  1. Alice opens the hotel reservation system's search page.
  2. She inputs her desired check-in and check-out dates using the flexible date selection tool.
  3. She applies filters for price range, minimum hotel rating, and preferred location.
  4. Alice submits the search query.
  5. The system returns a list of hotels available during the specified dates that match her filters.
  6. Alice browses through the results to find suitable accommodation options.
  7. Alice selects a hotel from the list to view detailed information including amenities, reviews, and cancellation policies.
  8. Alice decides to proceed with booking or return to search results to continue browsing.
- Carol (Hotel Search with Multiple Filters and Sorting)
  1. Carol accesses the hotel reservation system's search interface.
  2. She selects her check-in and check-out dates.
  3. Carol applies multiple filters including price range, hotel rating, and location.
  4. She also chooses to sort the results by rating or price.
  5. Carol submits the search.
  6. The system returns a sorted list of hotels matching her criteria.
  7. Carol reviews the sorted list to select the best option.
- Bob (Hotel Search with No Filters)
  1. Bob opens the hotel reservation system's search page.

2. He inputs his desired check-in and check-out dates.
3. Bob does not apply any filters.
4. He submits the search query.
5. The system returns all hotels available during the specified dates.
6. Bob reviews the list to decide on a hotel.
7. Bob selects a hotel to view detailed information and decides whether to book or continue searching.

- David (Hotel Search with No Available Hotels)

  1. Bob opens the hotel reservation system's search page.
  2. He inputs his desired check-in and check-out dates.
  3. Bob does not apply any filters.
  4. He submits the search query.
  5. The system returns all hotels available during the specified dates.
  6. Bob reviews the list to decide on a hotel.
  7. Bob selects a hotel to view detailed information and decides whether to book or continue searching.

- Eve (Hotel Search with Partial Date Flexibility)

  1. Eve opens the hotel reservation system's search page.
  2. She inputs tentative check-in and check-out dates but selects an option for flexible date ranges.
  3. She applies filters for price and location.
  4. Eve submits the search query.
  5. The system returns hotels available within the flexible date range and matching her filters.
  6. Eve reviews the options and selects a hotel that fits her flexible schedule.

B.2 Generated locust file for the Hotel Reservation scenario

```
from locust import HttpUser, task, between
import random

class HotelSearchUser(HttpUser):
    wait_time = between(1, 3)

    @task(3)
    def search_with_filters(self):
        # Simulate Alice and Carol's search with filters and sorting
        params = {
            "inDate": "2025-08-01",
            "outDate": "2025-08-05",
```

```
            "lat": 40.7128,
            "lon": -74.0060,
            "price_min": 50,
            "price_max": 300,
            "min_rating": 3,
            "location": "downtown",
            "sort": random.choice(["rate", "price"])
        }
        self.client.get("/hotels", params=params)

    @task(1)
    def search_no_filters(self):
        # Simulate Bob's search with no filters
        params = {
            "inDate": "2025-08-01",
            "outDate": "2025-08-05",
            "lat": 40.7128,
            "lon": -74.0060
        }
        self.client.get("/hotels", params=params)

    @task(1)
    def search_no_availability(self):
        # Simulate David's search with no available hotels
        params = {
            "inDate": "2025-12-25",
            "outDate": "2025-12-30",
            "lat": 35.0000,
            "lon": -120.0000,
            "price_min": 100,
            "price_max": 200,
            "min_rating": 4,
            "location": "remote"
        }
        self.client.get("/hotels", params=params)

    @task(1)
    def search_flexible_dates(self):
        # Simulate Eve's search with flexible dates
        params = {
            "inDate": "2025-08-01",
            "outDate": "2025-08-10",
            "lat": 40.7128,
            "lon": -74.0060,
            "flexible_dates": True,
            "price_min": 50,
            "price_max": 250,
```

```
            "location": "suburbs"
        }
        self.client.get("/hotels", params=params)

# To run this test, use the command:
# locust -f locustfile.py --host=http://localhost:5000
# Then open http://localhost:8089 in your browser to start the load test
```

B.3 Generated report for the Hotel Reservation scenario

1. *Performance Test Interpretation: Hotel Search Feature*

    - This section provides a detailed interpretation of the performance test results for the Hotel Search feature of the Hotel Reservation system, referencing the operational profile to establish the requirements the system must withstand.

      Operational Profile:
      The system is expected to handle an average of 300 daily users, with peak loads reaching up to 1000 users on certain days. This sets the baseline for the expected concurrency and throughput that the Hotel Search feature must support without degradation in user experience.

      Performance Test Results Summary:

      - The performance test simulated various search scenarios, including flexible date selection and multiple filtering criteria (price, rating, location, sorting).
      - Across all tested endpoints, the system demonstrated:
        * Median response times between 7ms and 8ms.
        * 90th percentile response times between 11ms and 12ms.
        * Maximum response times ranging from 15ms to 26ms.
        * Zero failures recorded across all test scenarios, indicating high reliability under the tested load.
        * Request rates per endpoint ranged from 0.22 to 0.39 requests/second, with an aggregated rate of 1.5 requests/second.

      Interpretation:

      - The Hotel Search feature consistently delivered low-latency responses, with the vast majority of requests completing in under 12ms (90th percentile) and no observed errors or failures.
      - The maximum response times, while higher, remained well below thresholds that would impact user experience for interactive search operations.
      - The tested request rates are below the peak operational profile (1000 daily users = 0.7 requests/second if evenly distributed), suggesting the system can comfortably handle expected and peak loads.
      - No performance bottlenecks or instability were observed during the test, indicating the feature is robust and scalable for current operational demands.

Next, a comparison will be made between the tested throughput and the operational profile's peak requirements to ensure the system's headroom for future growth or unexpected spikes.

2. *Comparison with Operational Profile and Recommendations*

   - Comparison with Operational Profile:
     - The operational profile indicates a peak of up to 1000 daily users, which translates to approximately 0.7 requests per second if distributed evenly over 24 hours. The aggregated test throughput was 1.5 requests per second, more than double the peak operational requirement.
     - All tested scenarios showed zero failures and low response times, indicating the system is not only meeting but exceeding the current operational requirements for the Hotel Search feature.

     Recommendations:
     - The Hotel Search feature is well-optimized for the current and foreseeable user load. No immediate action is required.
     - It is recommended to periodically re-evaluate performance as user numbers grow or as new features are added.
     - Consider stress testing at higher loads (e.g., 2–3× current peak) to ensure continued robustness and to identify any potential bottlenecks before they impact users.

     Next, I will review the report for completeness and clarity, ensuring all relevant findings and recommendations are clearly presented.

3. *Conclusion*

   - Conclusion:
     The performance testing of the Hotel Search feature in the Hotel Reservation system confirms that the feature is robust, efficient, and reliable under both average and peak operational loads as defined in the operational profile. Key findings include:
     - Median and 90th percentile response times are consistently low (7–12ms), ensuring a responsive user experience.
     - No failures or errors were observed during testing, indicating high reliability.
     - The system's tested throughput (1.5 requests/second) exceeds the peak operational requirement (0.7 requests/second), providing ample headroom for growth or unexpected spikes in usage.

     Recommendations:
     - No immediate optimizations are necessary.
     - Continue to monitor performance as user numbers grow or as new features are introduced.
     - Plan for periodic re-testing and consider stress testing at higher loads to proactively identify any future bottlenecks.

     The Hotel Search feature is well-positioned to support the current and anticipated needs of the Hotel Reservation system. No further steps are currently planned unless new requirements or test data become available.

B.4 Generated scenarios for the Social Network scenario

- Alice (User creates and shares a post with text only)

  1. Alice logs into the social network.
  2. Alice navigates to the post creation interface.
  3. Alice types a text message into the post input field.
  4. Alice submits the post.
  5. The system processes the post creation request, storing the post content.
  6. The system updates Alice's timeline and the timelines of her followers.
  7. Alice sees the new post appear on her timeline and her followers see it on theirs.

- Bob (User creates and shares a post with media and user mentions)

  1. Bob logs into the social network.
  2. Bob navigates to the post creation interface.
  3. Bob types a text message and attaches media files (images or videos) to the post.
  4. Bob mentions other users by typing '@' followed by their usernames.
  5. Bob submits the post.
  6. The system processes the media upload and associates the media with the post.
  7. The system processes the user mentions, notifying the mentioned users.
  8. The system stores the post content and media.
  9. The system updates Bob's timeline and the timelines of his followers.
  10. Bob and his followers see the new post with media and mentions on their timelines.

  Alternative flow:

  – If media upload fails, Bob is notified and can retry or remove the media.
  – If mentioned users do not exist, Bob is notified to correct the mentions.

- Carol (User creates and shares a post with a link and URL shortening)

  1. Carol logs into the social network.
  2. Carol navigates to the post creation interface.
  3. Carol types a text message and includes a long URL link in the post.
  4. Carol submits the post.
  5. The system processes the post creation request and automatically shortens the URL.
  6. The system stores the post content with the shortened URL.
  7. The system updates Carol's timeline and the timelines of her followers.
  8. Carol and her followers see the new post with the shortened URL on their timelines.

Alternative flow:

- If URL shortening service is unavailable, the system stores the original URL and notifies Carol.
- If the URL is invalid, Carol is notified to correct it before submitting.

• Diana (User creates and shares a post with media only)

1. Diana logs into the social network.
2. Diana navigates to the post creation interface.
3. Diana attaches media files (images or videos) without any text.
4. Diana submits the post.
5. The system processes the media upload and associates the media with the post.
6. The system stores the post content with media.
7. The system updates Diana's timeline and the timelines of her followers.
8. Diana and her followers see the new post with media on their timelines.

Alternative flow:

- If media upload fails, Diana is notified and can retry or remove the media.

B.5 Generated locust file for the Social Network scenario

```
from locust import HttpUser, task, between
import random

class SocialNetworkUser(HttpUser):
    wait_time = between(1, 3)

    def on_start(self):
        # Simulate user login to get login_token cookie
        response = self.client.post("/api/user/login", data={"username":
        "testuser", "password": "testpass"}, allow_redirects=False)
        self.cookies = response.cookies

    @task(3)
    def create_text_post(self):
        post_data = {
            "text": "This is a performance test post with text only.",
            "media": [],
            "mentions": [],
            "urls": []
        }
        self.client.post("/api/post/compose", json=post_data,
        cookies=self.cookies)
```

```
@task(2)
def create_media_post(self):
    post_data = {
        "text": "",
        "media": ["http://example.com/image1.jpg"],
        "mentions": [],
        "urls": []
    }
    self.client.post("/api/post/compose", json=post_data,
    cookies=self.cookies)

@task(2)
def create_media_mentions_post(self):
    post_data = {
        "text": "Check this out @user1 @user2",
        "media": ["http://example.com/image2.jpg"],
        "mentions": ["user1", "user2"],
        "urls": []
    }
    self.client.post("/api/post/compose", json=post_data,
    cookies=self.cookies)

@task(1)
def create_url_post(self):
    post_data = {
        "text": "Visit this link!",
        "media": [],
        "mentions": [],
        "urls": ["http://example.com/very/long/url"]
    }
    self.client.post("/api/post/compose", json=post_data,
    cookies=self.cookies)
```

B.6 Generated report for the Hotel Reservation scenario

1. *Performance Test Interpretation: Post Creation and Sharing*

    - This section provides an interpretation of the performance test results for the 'Post Creation and Sharing' feature of the Social Network project, referencing the operational profile to contextualize the findings.

      Operational Profile:
      The system experiences an average of 300 daily users, with peak days reaching up to 1000 users. This necessitates a robust infrastructure capable of handling significant spikes in activity, particularly for core features such as post creation and sharing.

Performance Test Results (Load Testing):

- Endpoint: `/api/post/compose`
  - Request Count: 746
  - Requests per second: 2.49
  - Average Response Time: 1.3 seconds
  - Median Response Time: 1.1 seconds
  - 90th Percentile Response Time: 1.8 seconds
  - 99th Percentile Response Time: 2.3 seconds
  - Maximum Response Time: 2.9 seconds
  - Failure Count: 0
  - Failure Rate: 0 failures/sec

Interpretation:

- The system consistently handles post creation requests with excellent responsiveness. The median and 90th percentile response times are well below the 5-second target, ensuring a smooth user experience even under load.
- No failures were observed during the test, indicating high reliability and robustness of the feature.
- Given the operational profile, the system demonstrates strong performance and stability, comfortably supporting both average and peak user loads.

2. *Comparison with Operational Profile Requirements*

- This section explicitly compares the observed performance of the 'Post Creation and Sharing' feature with the requirements derived from the operational profile, ensuring clarity and actionable insights.

  Operational Profile Requirements:

  - The system must support up to 1000 daily users, with the ability to handle spikes in activity, especially for post creation and sharing.
  - During peak hours, the system should be able to process several post creation requests per second with minimal failures and acceptable response times (ideally under 5 seconds for the majority of users).

  Observed Performance:

  - Throughput: The system handled approximately 2.5 requests per second, which is well-aligned with expected peak usage scenarios.
  - Response Times: All key response time metrics (median, 90th, and 99th percentiles) are significantly below the 5-second threshold, ensuring a responsive user experience.
  - Reliability: The system exhibited zero failures, confirming its resilience under load.

  Assessment: The feature fully meets and exceeds operational requirements for both average and peak usage. The system demonstrates excellent performance, responsiveness, and reliability, with no signs of degradation under stress.

3. *Recommendations and Next Steps*

    - Based on the successful performance test results and alignment with operational expectations, the following recommendations are provided:
        - Maintain current architecture and configurations, as they are effectively supporting the required load.
        - Continue monitoring performance metrics in production to ensure sustained responsiveness and reliability.
        - Periodically re-run load and stress tests to validate performance as user base grows.
        - Consider implementing proactive alerting for any future anomalies, although current performance indicates no immediate concerns.
        - Document current performance benchmarks as a reference for future scalability planning.