



Degree Project in Software Engineering of Distributed Systems

Second cycle, 30 credits

Development of an AI-driven DevOps Engineer: Automating Workflows with an LLM based Multi-Agent System

EUGEN LUCCHIARI HARTZ

Development of an AI-driven DevOps Engineer: Automating Workflows with an LLM based Multi-Agent System

EUGEN LUCCHIARI HARTZ

Master's Programme, Software Engineering of Distributed Systems, 120 credits
Date: July 1, 2025

Supervisors: Amirhossein Layegh Kheirabadi, Mohamed Nour Bouraoui
Examiner: Amir H. Payberah

School of Electrical Engineering and Computer Science

Host company: Knowit AB

Swedish title: Utveckling av en AI-driven DevOps-ingenjör: Automatisering av arbetsflöden med ett LLM-baserat multiagentsystem

Abstract

In today's fast-paced software engineering landscape, achieving reliable development and maintaining DevOps workflows has become increasingly challenging. These workflows often require manual configuration of pipelines, containerisation, and cloud deployment, which include repetitive, error-prone, and time-consuming tasks. This thesis investigates how Large Language Models (LLMs) and AI agents can be used to automate these workflows and addresses the challenge of creating an adaptive system that can interpret natural language input and automate complex DevOps tasks.

The study explores three system versions: a script-based system, a single-agent architecture with reasoning capabilities, and a modular multi-agent system including specialised agent roles for reasoning, prompt engineering, and error reflection. The systems were evaluated through a series of 12 representative DevOps tasks, where functionality, accuracy, efficiency, and reliability have been measured under controlled conditions. The results show that the multi-agent system achieved the highest task success rate (86.11%), significantly reduced execution times, and achieved a lower error rate.

This work demonstrates that incorporating ReAct-style reasoning, agent-based reasoning and modular design into DevOps automation not only enhances performance and adaptability but also maintains transparency and user oversight. The findings showcase the importance of human-in-the-loop control, where the user becomes a part of the automation process, for safety and trust in AI-driven automation. Overall, this thesis provides a foundation for developing an AI DevOps Engineer, including insights into system design, evaluation methodology, and the impact of prompt engineering.

Keywords

Large Language Models, AI Agents, Multi-Agent Systems, DevOps Automation, ReAct-style reasoning, Human-in-the-loop control, Prompt Engineering, Software Development, Cloud Deployment

Sammanfattning

I dagens snabbväxande landskap inom programvaruutveckling har det blivit alltmer utmanande att uppnå tillförlitlig utveckling och att upprätthålla DevOps-arbetsflöden. Dessa arbetsflöden kräver ofta manuell konfiguration av pipelines, containerisering och molndistribution, vilket innefattar repetitiva, felbenägna och tidskrävande uppgifter. Denna avhandling undersöker hur stora språkmodeller (LLMs) och AI-agenter kan användas för att automatisera dessa arbetsflöden och adresserar utmaningen att skapa ett adaptivt system som kan tolka naturligt språk och automatisera komplexa DevOps-uppgifter.

Studien utforskar tre systemversioner: ett skriptbaserat system, en enagentarkitektur med resonemangsförmåga samt ett modulärt multiagentsystem med specialiserade roller för resonemang, promptteknik och felreflektion. Systemen utvärderades genom en serie av 12 representativa DevOps-uppgifter, där funktionalitet, noggrannhet, effektivitet och tillförlitlighet mättes under kontrollerade förhållanden. Resultaten visar att multiagentsystemet uppnådde den högsta framgångsgraden för uppgifterna (86,11 %), signifikant minskade exekveringstiderna och uppnådde en lägre felfrekvens.

Detta arbete visar att införandet av ReAct-liknande resonemang, agentbaserat resonemang och modulär design i DevOps-automation inte bara förbättrar prestanda och anpassningsförmåga utan också bibehåller transparens och användarens kontroll. Resultaten belyser vikten av människa-i-loopstyrning, där användaren blir en del av automatiseringsprocessen, för att säkerställa säkerhet och förtroende för AI-drivna automationer. Sammantaget utgör denna avhandling en grund för utvecklingen av en AI DevOps-ingenjör och erbjuder insikter i systemdesign, utvärderingsmetodik och promptteknikens inverkan.

Nyckelord

Stora språkmodeller, AI-agenter, Multiagentsystem, DevOps-automation, ReAct-liknande resonemang, Människa-i-loopstyrning, Promptteknik, Programvaruutveckling, Molndistribution

Acknowledgments

- To my supervisor at KTH, Amirhossein Layegh Kheirabadi, who guided me through the entire thesis journey, provided invaluable feedback and had an answer to every question.
- To my examiner at KTH, Amir H. Payberah, who reviewed my work and offered insightful comments that helped me to improve the thesis.
- To Märta-Maria Jangenfalk and Mohamed Nour Bouraoui at Knowit Connectivity, who always supported and encouraged me throughout my Master's thesis project and created a very motivating and collaborative environment.
- To my colleagues and friends at Knowit Connectivity, who were always ready to assist with technical questions, share ideas, and contribute to an inspiring work atmosphere.
- To the entire Knowit Connectivity team for creating a supportive and innovative culture where everyone helps each other, making it a pleasure to work and learn.
- To friends and family who never forget and always support me during my stay in the far north in Stockholm, Sweden.

Stockholm, July 2025

Eugen Lucchiari Hartz

Contents

1	Introduction	1
1.1	Background	1
1.2	Problem	3
1.3	Purpose	3
1.4	Goals	3
1.5	Research Methodology	4
1.6	Delimitations	5
1.7	Structure of the Thesis	6
2	Background	7
2.1	DevOps Workflows	7
2.2	Automation in DevOps	8
2.3	Large Language Models for Code Generation	9
2.3.1	Prompt Engineering	10
2.4	AI Agents for DevOps Automation	10
2.4.1	Autonomous AI Agents	11
2.4.2	ReAct-style Reasoning (Thought → Action → Result)	12
2.4.3	Multi-Agent Architectures and Coordination	13
2.5	Related Work	13
2.6	Summary	14
3	Methodology and System Development	17
3.1	Research Process	17
3.2	Overview of the Systems	18
3.3	System Architecture and Design	19
3.3.1	Version 1 - Script-Based Automation Framework	22
3.3.2	Version 2 – Single-Agent AI DevOps Engineer	26
3.3.3	Version 3 – Multi-Agent AI DevOps Engineer	33
3.4	Evaluation Approach	42

3.5	Societal and Ethical Considerations	45
4	Results and Analysis	47
4.1	Functional Evaluation	47
4.2	Efficiency Analysis	49
4.3	Reliability Analysis	50
4.4	User Experience	51
4.5	Interpretation of Results	52
5	Conclusion and Future Work	55
5.1	Contributions	55
5.2	Future Work and Extensions	56
5.3	Limitations	57
5.4	Final Reflections	58
	References	59
A	GitHub Repositories	71
B	Test Evaluation Table	73

List of Figures

3.1	Example of the User Interface (UI) of system version 3. However, all three systems have a similar UI.	21
3.2	Interaction flow in the script-based system.	24
3.3	Interaction flow in the single-agent system.	28
3.4	The frontend displaying the AI agent's reasoning during execution, following the Thought → Action → Result loop. . .	31
3.5	UI showing human-in-the-loop control in action as the system is waiting for user approval. Users can approve, edit, or reject the suggested command.	32
3.6	Edit mode allows users to adjust AI-generated commands before execution.	33
3.7	Fullscreen view of the UI in Version 3, showing the full Thought → Action → Result loop during task execution. The layout improves clarity and user experience.	35
3.8	Interaction flow in the multi-agent system.	36
3.9	System execution workflow for the example task: "Create a GitHub Actions pipeline.": Shows how the user input is processed and executed through coordinated agent interaction, resulting in the creation and deployment of <code>.github/workflows/workflow.yml</code>	40
4.1	Task Success Rate per System Version: Percentage of tasks each system completed successfully without requiring manual correction.	48
4.2	Tasks Attempted vs. Successfully Completed: Number of tasks each system attempted and how many were completed successfully.	49
4.3	Average Execution Time per Task: Compares the average time taken to complete a task across the three system versions. . . .	50

4.4	Execution Recovery Rate Across System Versions: Shows how reliably each system recovered from execution errors. Note: v3 had only 2 recovery opportunities, making its recovery rate less statistically representative than v2.	51
B.1	Test Evaluation Table - Part 1: Tasks 1–3	73
B.2	Test Evaluation Table - Part 2: Tasks 4–6	73
B.3	Test Evaluation Table - Part 3: Tasks 7–9	74
B.4	Test Evaluation Table - Part 4: Tasks 10–12	74

List of acronyms and abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
AWS	Amazon Web Services
CI/CD	Continuous Integration and Continuous Deployment
DevOps	Development and Operations
DSR	Design Science Research
LLM	Large Language Model
RAG	Retrieval-Augmented Generation
ReAct	Reasoning and Acting
RQ	Research Question
UI	User Interface
UX	User Experience
YAML	YAML Ain't Markup Language

Chapter 1

Introduction

1.1 Background

Software systems nowadays are built and maintained in environments that require speed, reliability and frequent updates. Therefore, **Development and Operations (DevOps)** has become a key approach in modern software engineering. The term combines development (Dev) and operations (Ops) and describes a way of working that focuses on collaboration, automation and continuous delivery across the entire software lifecycle from planning to deployment [1].

A **DevOps** workflow puts this approach into action by defining the step-by-step processes that teams follow to build, test and release software efficiently [2]. A common example is a **Continuous Integration and Continuous Deployment (CI/CD)** pipeline, where code changes are automatically tested and deployed whenever they are pushed to the repository [3, 4]. Other examples are the containerisation of an application using tools like Docker and cloud deployment using platforms such as **Amazon Web Services (AWS)**, Azure or Google Cloud.

While **DevOps** workflows aim to increase automation, setting up and maintaining these workflows still often involves manual, repetitive and error-prone tasks [5]. Developers have to write detailed configuration files, manage integration between various tools and update automation scripts to keep up with evolving project needs. This manual effort can lead to inconsistencies, slow down the delivery of software and increase the risk of deployment failures [6].

Recent advancements in **Artificial Intelligence (AI)** and specifically in **Large Language Models (LLMs)** [7] have opened new opportunities to

improve this process. LLMs are capable of understanding natural language and translating it into technical code, scripts or configurations [8, 9]. For example, an LLM can take the instruction “Create a GitHub Actions pipeline for my app” and produce a complete **YAML Ain’t Markup Language (YAML)** workflow file. This ability of LLMs to generate technical code based on user inputs provides a huge potential to reduce manual intervention in software deployment processes. It therefore lowers the barrier to automation and enables developers to focus on higher-level goals rather than low-level scripting.

However, LLMs alone are reactive systems. They generate outputs when prompted but do not autonomously plan, execute or react to actions based on results [10, 11]. To create a higher level of automation, LLMs can be integrated into AI agents [11]. AI agents are systems that reason about tasks, make decisions and interact with their environment. They extend the capabilities of LLMs by providing a structured loop of reasoning, acting and observing results which enables more autonomous and intelligent behaviour [12, 13].

Combining LLMs and AI Agents therefore can create a possibility to efficiently and intelligently automate DevOps workflows such as the generation of a CI/CD pipeline, the containerisation of an application or the deployment of an application to the cloud [14, 15].

This thesis investigates this potential by developing and evaluating three system versions:

- A script-based automation framework that connects user input to predefined hardcoded workflows.
- A single-agent system that uses Reasoning and Acting (ReAct)-style reasoning (Thought → Action → Result) [16] and user-in-the-loop control [17] to interpret user goals and plan automation steps dynamically.
- A multi-agent system that additionally coordinates specialised agents for reasoning, prompt engineering, and error handling [18], enabling more modular, transparent, and reliable automation.

The study introduces core concepts such as ReAct reasoning, agent coordination, and user-in-the-loop control, which are essential for achieving safe, effective and transparent AI-driven automation.

1.2 Problem

As outlined in the background, setting up and maintaining **DevOps** workflows such as **CI/CD** pipelines, containerisation, and cloud deployment often remains a manual process. This involves repetitive and error-prone tasks like writing configuration files, integrating diverse tools, and continually updating scripts to accommodate evolving project requirements [5, 6].

Those aspects lead to the following **Research Question (RQ)** and Hypothesis of the thesis:

- **RQ:** How can **LLMs** and **AI** agents be used to automate **DevOps** workflows through natural language while ensuring reliability, transparency and user control in the automation process?
- **Hypothesis:** A system combining **LLMs** and **AI** agents with **ReAct**-style reasoning and user-in-the-loop control can automate **DevOps** workflows more efficiently and reliably than script based systems, single agent systems or manual approaches.

1.3 Purpose

The main purpose of this study is to explore how to reduce manual effort and human error in **DevOps** workflows by introducing **AI**-driven automation. Besides this practical aspect, the study also contributes academically by exploring to what extent **LLMs** and **AI** agents can be applied to real-world **DevOps** tasks. Additionally, the project emphasises ethical and societal considerations by designing automation that remains transparent, controllable, and centred around human oversight.

1.4 Goals

The main goal of this thesis is to develop a system using **LLMs** and **AI** agents to automate key **DevOps** workflows such as **CI/CD** pipeline creation, application containerisation and cloud deployment based on natural language input.

Several subgoals further define the scope and contribution of the project:

- **Increase efficiency and accessibility in **DevOps** Workflows:** Automate common **DevOps** tasks to reduce manual effort, minimise human errors and make automation more accessible for developers.

- **Leverage LLMs and AI Agents with ReAct-style Reasoning:** Apply LLMs and AI agents using ReAct-style reasoning and modular multi-agent coordination to enable intelligent, adaptive and transparent DevOps automation.
- **Enhance Transparency, User Control, and Trust:** Integrate human-in-the-loop control to provide transparency in decision-making, allow user intervention, and increase trust in AI-driven automation systems.
- **Demonstrate Engineering Competence:** Design, implement, and evaluate a functioning AI-driven automation system to fulfil the Master's thesis requirements and demonstrate practical engineering skills.

1.5 Research Methodology

This thesis follows a Design Science Research (DSR) methodology, which is well-suited for projects focused on developing and evaluating technical systems. The goal of DSR is to create innovative solutions to real-world problems while also contributing practical knowledge to the field. The DSR process includes six key stages: (1) Problem Identification and Motivation, (2) Define Objectives for a Solution, (3) Design and Development, (4) Demonstration, (5) Evaluation, and (6) Communication [19, 20, 21]. In this case, the project focuses on building an AI-based system for automating DevOps workflows using LLMs and AI agents.

The research is based on a pragmatic philosophical approach as it aims to solve a real-world engineering challenge while generating insights that can be applied beyond this specific project. This aligns with the dual objective of the thesis: Delivering a working system and gaining a deeper understanding of how LLMs and AI agents can be effectively and reliably used in DevOps automation.

The research approach is engineering-driven and iterative. The system was developed through three stages: Script-based, single-agent, and multi-agent versions where each version is building on the previous one. After each stage, the system was evaluated and refined based on observed results so that there is an incremental improvement process.

Other research methods were also considered but not selected. A case study [22] was rejected because the focus is on system development rather than analysing an organisational setting. An experiment-only approach [23] is also

insufficient, as it would not capture the evolving design and implementation work.

The evaluation of the system uses both quantitative and qualitative methods. Quantitative evaluation includes measurements such as execution time and execution success rates. The qualitative evaluation focuses on reasoning trace analysis, system reliability and the security of the automation process. The detailed application of this methodology is presented in Section 3.1.

1.6 Delimitations

This thesis is limited in scope to ensure a focused investigation within the available time frame and resources. While the developed system demonstrates the potential of using LLMs and AI agents for DevOps automation, the following delimitations have been set for this thesis:

- **Limited task coverage and validation:** The system focuses on automating selected DevOps tasks and scenarios and is evaluated on predefined tools and scenarios such as CI/CD pipelines, Docker containerisation and cloud deployment. It does not cover the full range of DevOps workflows or tools used in production environments and compatibility with other workflows or tools has not been evaluated.
- **No in-depth LLM comparison, fine-tuning or Retrieval-Augmented Generation (RAG):** The thesis relies on limited exploration of Claude Sonnet 3.5 [24]. During development, the system relied on a single pre-trained LLM accessed via Ollama [25] (first Deepseek Coder v2 [26], later Qwen 2.5 Coder [27]) and Claude Sonnet 3.5 was used for evaluation. It does not include detailed benchmarking across different models, apply fine-tuning techniques, or incorporate RAG methods.
- **No persistent agent memory:** The system does not include memory across sessions. Each user interaction is handled independently, without long-term learning or adaptation.
- **Evaluation based on predefined scenarios:** The system is evaluated using a set of controlled, predefined use cases. There is no large-scale user study or deployment in real-world production settings.

- **No in-depth User Experience (UX) validation:** While interaction flows are analysed qualitatively, the project does not include formal UX testing or usability evaluation.

These boundaries have been defined to ensure having a clear defined scope of the project.

1.7 Structure of the Thesis

The thesis is organised into five main chapters, followed by appendices containing supporting materials.

Chapter 2 provides the necessary theoretical background on DevOps workflows, LLMs, AI agents, ReAct-style reasoning, multi-agent architectures, and related work. It builds the foundation needed to understand the context and motivation for the system development.

Chapter 3 describes the research methodology and outlines the research process followed throughout the project. It presents the system architecture and design, details about the development of the three system versions (script-based, single-agent, and multi-agent), explains the evaluation strategy, and discusses relevant societal and ethical considerations.

Chapter 4 presents the results and analysis of the system evaluation. It includes the functional evaluation, efficiency analysis, reliability analysis, and insights into UX. Furthermore, it interprets the results in relation to the research goals set out at the beginning of the thesis.

Chapter 5 summarises the main contributions of the work, discusses its limitations, outlines possible directions for future research, and provides final reflections on the project's outcomes and societal implications.

Finally, the appendices contain supporting materials including the links to the GitHub repositories of the implemented systems and the evaluation table containing evaluation documentation of all three system versions.

This structure aims to guide the reader from theoretical foundations to practical implementation and evaluation.

Chapter 2

Background

This chapter provides the theoretical foundation for the thesis by presenting key concepts and technologies relevant to the development of an **AI-driven DevOps** automation system. It begins with an overview of **DevOps** workflows and their components, followed by a discussion of automation tools, **LLMs**, and **AI** agents. The chapter also introduces reasoning strategies such as **ReAct**-style reasoning and multi-agent architectures, and concludes with a review of related work. These concepts together form the basis for understanding the motivation, design choices, and evaluation criteria used throughout the project.

2.1 DevOps Workflows

DevOps is a set of practices that integrates software development (Dev) and IT operations (Ops) to enable faster, more reliable, and more frequent delivery of software [1]. Central to this approach is the automation of workflows that support the full software lifecycle - from coding and integration to testing, deployment, and infrastructure management.

A typical **DevOps** workflow includes several interconnected components. **CI/CD** pipelines for example automatically build, test, and deploy code changes whenever updates are pushed to a version control system such as Git. These pipelines improve consistency and reduce the risk of human error during releases. Version control ensures traceability and collaboration, while automated testing provides immediate feedback on the quality and stability of new code [2].

Another key component is containerisation, often implemented using tools like Docker or Kubernetes. Containers package applications and their

dependencies into isolated, reproducible units that can run consistently across different environments. This isolation reduces configuration conflicts between systems and simplifies testing and deployment. Containers also make it easier to scale applications horizontally and integrate them into cloud-native architectures such as micro services [28, 29].

For deploying applications at scale, many organisations rely on cloud infrastructure, which is often managed through Infrastructure as Code (IaC) tools such as Pulumi or Terraform. These tools enable the declarative definition of infrastructure and simplify provisioning of cloud resources on platforms like AWS, Azure, or Google Cloud [30, 31].

While these tools offer powerful capabilities, setting up DevOps workflows still often requires writing complex configuration files, connecting multiple services, and debugging deployment issues. The process can be time-consuming, repetitive, and prone to error, especially when adapting workflows for new projects or teams [5, 6]. These challenges highlight the need for more intelligent and flexible automation solutions, an area in which Large Language Models and AI agents offer significant potential [15].

2.2 Automation in DevOps

Automation is a fundamental principle in DevOps, aiming to reduce manual effort, improve reliability, and enable rapid delivery cycles. To support this, a wide range of tools has emerged to automate various stages of the software development and deployment process. Among the most widely adopted are GitHub Actions and GitLab CI/CD, which allow teams to define custom pipelines for building, testing, and deploying applications [32, 33, 34].

These tools enable developers to write pipeline configurations - typically in YAML format - that specify a sequence of steps triggered by events such as code pushes, pull requests, or scheduled timers. Common tasks include compiling source code, running unit and integration tests, building Docker containers, and deploying to staging or production environments. By embedding automation directly into the version control workflow, these systems help ensure consistency and reduce the likelihood of human error [34].

Despite their benefits, these tools still require a considerable amount of manual configuration and maintenance. Developers must be familiar with the syntax, understand the structure of CI/CD files, and integrate various third-party services or scripts. As project complexity grows, keeping pipeline logic correct, efficient, and up to date can become a significant burden. This limits accessibility, especially for smaller teams or less experienced developers and

introduces room for misconfiguration [33].

These challenges motivate the need for more intelligent and adaptive forms of automation. Recent advances in **LLMs** and **AI** agents [15] present an opportunity to simplify **DevOps** workflows by enabling systems that can understand developer intent expressed in natural language, generate appropriate automation logic, and even reason through multi-step tasks. The following sections explore these technologies in more detail and examine how they can be applied to **DevOps** automation.

2.3 Large Language Models for Code Generation

LLMs have become powerful tools for automating software development tasks, including code generation, configuration file creation, and script writing [8, 9]. Models such as GPT, DeepSeek Coder, Claude, and Qwen are trained on vast amounts of data and code, enabling them to understand natural language instructions and produce syntactically correct and contextually relevant outputs. This capability allows developers to describe their intent in plain language, for example, “create a GitHub Actions workflow for a Node.js project”, and receive a functional automation script in return [35, 36, 37].

The benefits of **LLMs** in the context of **DevOps** are substantial. They can significantly reduce manual effort, accelerate onboarding for new team members, and make automation more accessible to those without deep technical knowledge. By translating human-readable input into executable code [37], **LLMs** have the potential to democratise access to **DevOps** tooling and lower the barrier to entry for configuring complex workflows while at the same time highly boost the development speed.

However, **LLMs** are not without limitations. A major concern is the risk of hallucination, where the model generates outputs that appear correct but are logically flawed, insecure, or entirely fabricated. In the context of **DevOps**, such issues could lead to incorrect deployments, broken workflows, or potential security vulnerabilities. Therefore, outputs must be carefully validated, especially when deployed in production environments [38, 39, 40].

To improve the reliability and relevance of generated outputs, prompt engineering plays a critical role. By structuring prompts carefully and providing relevant context, such as repository content, the quality of **LLM** responses can be significantly improved. This aspect will be examined in more detail in the next section.

2.3.1 Prompt Engineering

Prompt engineering refers to the practice of crafting inputs to **LLMs** in a way that guides them towards generating accurate, relevant, and contextually appropriate outputs. In systems that rely on **LLMs** for **DevOps** automation, prompt engineering is not just an auxiliary detail - it is critical to the system's stability, correctness, and reliability [41, 42].

A prompt is the input given to an **LLM** to instruct it on what task to perform. It can be a natural language question, command, or structured example, and directly influences the model's output [41].

Well-designed prompts can significantly reduce the likelihood of hallucinated or faulty outputs by clearly defining the task, constraints, and expected format. Their impact on an **LLM**'s output is therefore highly significant. In the context of **DevOps**, where generated outputs may directly affect code repositories, pipelines, or deployment environments, even small errors can lead to system failures or security issues. Prompt engineering serves as the first layer of control to prevent such outcomes [39].

To enhance accuracy, prompts can be augmented with dynamic context, such as repository structure, existing configuration files, or recent execution history. This allows the model to generate solutions that are not only technically correct but also well-adapted to the specific project or environment. System-level instructions and structured formatting (e.g., few-shot examples or templates) can also improve the consistency and predictability of responses [43, 44].

However, prompt engineering also comes with challenges. One is the inherent ambiguity of natural language. Users may describe their goals in vague or incomplete terms, leaving room for misinterpretation. Another aspect is the trade-off between specificity and flexibility: overly detailed prompts may overfit to narrow use cases, while overly general prompts can result in incorrect or irrelevant outputs [45].

Managing these trade-offs is a central concern in the design of **LLM**-driven systems. In this project, prompt engineering is treated as a key design component and is iteratively refined to improve the system's overall accuracy, robustness, and user alignment.

2.4 AI Agents for DevOps Automation

While **LLMs** are powerful tools for translating natural language into technical code and scripts, they are only reactive systems which means that they

produce outputs in response to prompts but do not independently plan or carry out multi-step processes. To enable more autonomous and goal-directed behaviour, **LLMs** can be embedded within **AI** agents [16].

AI agents are systems designed to interpret user input, reason about goals, and execute actions in a structured and sequential manner. In the context of **DevOps** automation, agents play a central role in bridging the gap between a developer's high-level intent and the technical execution of tasks such as cloning repositories, generating workflow files, building containers, and deploying applications [46].

By wrapping **LLMs** inside agents, it becomes possible to build interactive systems that do more than generate static outputs. Agents can analyse context, make decisions based on past steps, and adapt dynamically to the outcome of executed actions. For example, if an error occurs during an execution step, the agent can reflect on the failure, revise its strategy, and attempt an alternative action [13, 47].

This structured, goal-oriented design transforms the **LLM** from a passive reactive assistant into an autonomous collaborator which is capable of managing complex **DevOps** workflows step by step. The following subsections introduce key architectural patterns used in this thesis to enable such behaviour: autonomous agents, **ReAct**-style reasoning, and multi-agent coordination.

2.4.1 Autonomous AI Agents

Autonomous **AI** agents are systems capable of operating with a defined goal, maintaining an internal representation of state, and interacting with their environment to complete tasks. Unlike traditional scripts that follow a fixed sequence of instructions, agents are designed to plan, adapt, and respond dynamically based on context and observed outcomes [12, 48].

In the context of **DevOps** automation, autonomous agents can execute sequences of commands such as cloning a repository, generating a workflow file, or building and deploying a container while reasoning about each step and adjusting their behaviour when necessary. This ability to track progress, manage dependencies, and recover from failures enables a more robust and intelligent automation process [49].

The key distinction between autonomous agents and static scripts lies in flexibility and decision-making. A script will run as written, regardless of environmental changes or intermediate failures. In contrast, an agent can evaluate the success of each action, revise its plan, and choose alternative

strategies if required [50, 51].

This autonomy forms the foundation for building an **AI** driven **DevOps** Engineer that is more resilient, adaptable, and capable of handling complex workflows. It also enables higher-level features such as feedback loops, context awareness, and goal-directed reasoning which are core building blocks of the system developed and evaluated in this thesis.

2.4.2 ReAct-style Reasoning (Thought → Action → Result)

ReAct-style reasoning is an approach to agent decision-making that breaks down complex problem-solving into a transparent and iterative cycle of Thought → Action → Result (or Observation). In this pattern, the agent first reflects on the current situation (Thought), then performs a specific operation (Action), and finally evaluates the outcome (Result or Observation). This structure allows agents to reason step by step and adapt their strategy based on feedback from the environment [16, 52, 53].

In the context of **DevOps** automation, this reasoning loop enables agents to dynamically plan and adjust execution flows as they progress through multi-step tasks. For instance, if an agent attempts to build a Docker container and the operation fails, the resulting observation can be used to inform the agent about it so that it can accordingly think of an alternative solution, for example modifying the Dockerfile or reattempting with different parameters. This allows the system to behave more intelligently than a static predefined script, which would simply fail without recovery [14, 54].

One of the key benefits of the **ReAct** pattern is its transparency and interpretability. By making each reasoning step explicit, it becomes easier to understand why the agent made a certain decision. This transparency is very important as trust, safety and security are crucial aspects to consider when it comes to automation with **AI**. Developers and users can follow the logic of the agent, which also makes it easier to debug, audit, and validate the agents behaviour [16, 55, 56, 46].

Moreover, the **ReAct** loop provides a structure for user-in-the-loop control. In this developed system during the thesis, each Thought → Action → Result cycle is exposed to the user through the **User Interface (UI)**, allowing manual approval, editing, or rejection of commands before they are executed. This added layer of control improves safety and trust in **AI**-driven automation and allows users to intervene when needed, without interrupting the agent's overall flow [57, 58, 59].

ReAct-style reasoning therefore offers a flexible and robust framework for building intelligent, controllable **AI** agents that can safely and trustworthily automate complex **DevOps** workflows.

2.4.3 Multi-Agent Architectures and Coordination

In the context of intelligent automation, multi-agent architectures enable a separation of concerns, where specific functions such as reasoning, prompt construction, validation, or error handling can be delegated to dedicated agents. This modular structure improves the maintainability, scalability, and robustness of the system. It allows for easier debugging, better error recovery, and makes the system more modular [18, 60].

There are various models of agent coordination, such as a centralised orchestration, where a master controller assigns tasks to specialised agents or decentralised approaches, where agents interact more autonomously and negotiate actions. When developing an automating system that requires the execution of multiple steps, centralised coordination is commonly used to maintain a coherent step-by-step process while still leveraging the specialisation of individual agents [61, 62, 63].

The main advantage of a multi-agent approach is its resilience and adaptability. If one agent fails or produces an unsatisfactory result, another agent can intervene, either to propose an alternative or to refine the process. This creates opportunities for fallback, recovery, and more reasonable decision-making [64].

Multi-agent coordination is increasingly relevant as automation systems grow in complexity, requiring transparent reasoning, adaptive behaviour, and human oversight. It provides a foundation for building intelligent systems that are not only capable of executing tasks, but also of reasoning about them and adjusting their behaviour in real time [63, 65, 66, 59].

2.5 Related Work

DevOps automation has traditionally relied on tools such as GitHub Actions, GitLab **CI/CD**, Terraform, and Docker Compose, which allow developers to define and automate build, test, and deployment workflows. While these tools are powerful and widely used, they depend on manual configuration and scripting. As projects grow in complexity, maintaining and extending such automation becomes increasingly error-prone and time-consuming [67, 5, 68].

In parallel, the emergence of **LLMs**, such as GPT, Claude Sonnet, DeepSeek Coder, and Qwen Coder, has opened up new possibilities for code generation and developer assistance. These models can produce working scripts, configuration files, and documentation from natural language input, making them useful for automating parts of the software development lifecycle. However, **LLMs** in isolation lack the ability to plan or adapt beyond single-step responses [69, 70, 71].

To address this limitation, researchers and developers have started integrating **LLMs** into agent-based systems. Early examples include LangChain agents [72], which use reasoning chains to perform multi-step tasks, and systems like the OpenAI Operator [73], which combine **LLMs** with predefined tool usage to execute tasks. These systems mark a step toward more autonomous software assistants but are still relatively constrained in scope, often lacking dynamic fallback mechanisms or user-in-the-loop transparency [74, 75].

This thesis builds upon several of these ideas by combining prompt-based code generation, **ReAct**-style reasoning, user-in-the-loop control, and multi-agent coordination into a unified **DevOps** automation system. It investigates how **LLMs** and **AI** agents can be integrated not just to generate code, but to plan, act, and adapt within complex workflows while involving the user in key decision points [60, 16, 76].

Unlike some related work, this project does not attempt to cover production-grade infrastructure automation using tools like Terraform or advanced cloud orchestration platforms. These areas involve additional challenges related to security, cost, and risk, and are considered outside the scope of this thesis.

2.6 Summary

This chapter has introduced the theoretical foundations relevant to the development of an **AI-driven DevOps** automation system. First it outlined key concepts in **DevOps** workflows, including **CI/CD** pipelines, containerisation, and cloud provisioning, and highlighted the challenges associated with their manual configuration and maintenance.

The discussion then turned to existing automation tools such as GitHub Actions and GitLab **CI/CD**, acknowledging their strengths while identifying limitations that motivate the need for more intelligent, adaptive solutions. **LLMs** were introduced as a promising technology for automating code and script generation from natural language input, along with the crucial role of

prompt engineering in guiding their output effectively.

Building on this, the chapter explored how **AI** agents can extend the capabilities of **LLMs** by enabling autonomous, goal-directed behaviour. The **ReAct** reasoning pattern was presented as a transparent and flexible decision-making loop, well-suited for interactive automation systems that provide user-in-the-loop control. Finally, the concept of multi-agent architectures was introduced as a modular, scalable approach to coordination, recovery, and system resilience.

These concepts form the basis of the system developed in this thesis, which is presented in the next chapter along with the applied methodology, design choices, and implementation process.

Chapter 3

Methodology and System Development

This chapter describes the methodology and engineering process used to design, implement, and evaluate the **AI-driven DevOps** automation system developed in this thesis. It begins with an overview of the research strategy, followed by a step-by-step explanation of the development process. The chapter then presents the overall system architecture and the evolution of three system versions: script-based, single-agent, and multi-agent. It concludes with a description of the evaluation strategy and a reflection on the societal and ethical implications of the system.

3.1 Research Process

The research process for this thesis follows as mentioned in Section 1.5 the **DSR** methodology, which is well suited for projects focused on developing and evaluating technical systems intended to solve real-world problems. The **DSR** approach emphasises a close connection between the engineering of a solution and the generation of applicable knowledge. In this thesis, the focus is on building an **AI-based** system for **DevOps** automation using **LLMs** and **AI** agents [21].

The process was carried out in an iterative build-test-refine loop, in which the system was incrementally developed through a series of design cycles. Each iteration aimed to enhance the system's capabilities while simultaneously evaluating its performance and identifying areas for improvement.

The research process included the following steps:

1. **Problem Identification:** Recognising that manual **DevOps** processes,

such as **CI/CD** setup, containerisation, and cloud deployment, are time-consuming, repetitive, and prone to human error.

2. **Review of Existing Work:** Studying the current landscape of **DevOps** automation tools, **LLM**-based code generation systems, and agent-based **AI** architectures to establish a foundation for system design.
3. **System Design and Development:** Creating three system versions with increasing levels of intelligence and modularity: a script-based baseline, a single-agent version with **ReAct** reasoning and human-in-the-loop control, and a multi-agent version with additionally coordinated agents.
4. **Application to Realistic Scenarios:** Implementing and testing the system on practical **DevOps** use cases, including GitHub Actions workflow generation, Docker-based containerisation, Kubernetes clustering and Cloud deployment.
5. **Evaluation:** Assessing each version of the system through both quantitative metrics (e.g., execution time, task success rate) and qualitative analysis (e.g., reasoning trace evaluation, user interaction flow).
6. **Reflection and Refinement:** Using evaluation insights to improve the system architecture, prompt engineering strategies, and agent coordination mechanisms across iterations.

This process allowed for continuous improvement of the system while aligning engineering outcomes with academic goals.

3.2 Overview of the Systems

To explore the potential of **LLMs** and **AI** agents for intelligent **DevOps** automation, the system developed in this thesis evolved through three distinct versions, each reflecting a different level of complexity, autonomy, and architectural sophistication. These versions were developed iteratively using the **DSR** methodology and serve as both working systems and as a basis for evaluating the core **RQ**.

The first version, the Script-Based Automation Framework, was designed to validate the feasibility of automating **DevOps** tasks through user input. In this version, specific scenarios such as **CI/CD** pipeline generation, Docker containerisation, Kubernetes clustering and simulated **AWS** deployment were

implemented using fixed Python scripts. Although limited in flexibility, this version established a working backend-frontend structure and provided valuable insights into the practical requirements and challenges of automating common **DevOps** workflows.

The second version, the Single-Agent **AI DevOps** Engineer, replaced the hardcoded logic from version 1 with a **ReAct**-style **LLM**-driven **AI** agent. This agent was capable of interpreting natural language input, planning **DevOps** actions, and executing them in a dynamic Thought → Action → Result loop. It also introduced human-in-the-loop control, allowing users to review, edit, approve or reject each proposed action before execution. This inclusion of the user increased system transparency and user trust, while also reducing the risk of executing incorrect or unsafe commands. Overall, this version significantly increased automation capabilities and flexibility, enabling the system to handle a broader range of tasks beyond predefined scenarios.

The third and final version, the Multi-Agent **AI DevOps** Engineer, has similar features as version 2 but additionally introduced a modular architecture composed of specialised agents, including a Reasoning Agent, Prompt Agent, and Reflector Agent, coordinated by an Agent Orchestrator. This design allowed for better separation of concerns, targeted behaviour, and improved error recovery, resulting in a system that is more maintainable, resilient, and scalable.

Each version built upon the previous one, not only expanding functionality but also deepening the system's reasoning capabilities and enabling user-in-the-loop control. Together, the three versions form a progression that reflects the thesis goal: to develop an increasingly intelligent and reliable automation framework that leverages both **LLMs** and structured **AI** agents for real-world **DevOps** scenarios.

3.3 System Architecture and Design

The **AI**-driven **DevOps** Engineer developed in this thesis is composed of modular components that work together to interpret user input, generate appropriate automation logic using an **LLM**, and execute **DevOps** tasks across various scenarios. The system architecture follows a client-server model [77], with a web-based ReactJS frontend [78], a FastAPI-powered backend [79], and an integrated **LLM** layer. The design emphasises modularity, transparency, and user-in-the-loop control [57], consistent with the principles of the **DSR** methodology [21].

The frontend is implemented using ReactJS and provides the **UI** for

entering natural language instructions, providing the repository name, reviewing **AI**-generated reasoning steps, approving, editing or rejecting actions, and viewing real-time execution logs (see Figure 3.1 for an example of the **UI** which looks similar for all three system versions). It uses Tailwind Cascading Style Sheets (CSS) for styling [80], Headless **UI** for accessibility and interaction components [81], and Framer Motion to provide smooth visual **UI** transitions [82].

Eugenius Multi Agent Devops Engineer

Describe what you want (e.g., 'Create a GitHub Actions pipeline')

Enter your GitHub repository name

Execute

Cancel

⌚ Total Execution Time: 30 seconds

⏸ Awaiting user approval...

📄 Result: ✅ Successfully executed: git clone https://github.com/eugenius0/github-actions-auto.git

💡 Thought: Since the repository has been successfully cloned, the task is complete

Final Answer: The GitHub repository 'github-actions-auto' has been successfully cloned using HTTPS authentication method from https://github.com/eugenius0/github-actions-auto.git. No further actions are required.

🎉 All steps executed.

✅ Task Completed!

Execution History

10/06/2025, 13:58:26

Completed 🤖 AI DevOps Agent completed the requested automation

⌚ 30 seconds

10/06/2025, 13:57:31

Completed 🤖 AI DevOps Agent completed the requested automation

⌚ 30 seconds

Figure 3.1: Example of the UI of system version 3. However, all three systems have a similar UI.

The backend is built with FastAPI, a lightweight asynchronous Python framework. It is responsible for handling **Application Programming Interface (API)** routing, coordinating agent logic, managing task execution, and serving the reasoning loop to the frontend. The backend acts as the central processing unit of the system, orchestrating interactions between the frontend, the agents, and the **LLM**.

The **LLM** layer uses Claude Sonnet 3.5 (before Qwen 2.5 Coder and DeepSeek Coder v2, served locally via Ollama), to interpret natural language input and generate corresponding commands, scripts, or configuration files. The integration supports context-aware prompting and plays a critical role in enabling dynamic and flexible automation.

The system is implemented using two primary programming languages: Python for the backend and agent logic, and TypeScript for the frontend. **DevOps** tools supported by the system include GitHub, GitLab, Docker, Kubernetes, and **AWS**, with automation scenarios tailored to workflows common to these platforms.

The overall development approach was incremental and modular. Each component was developed and tested individually before being integrated into the complete system, allowing for flexibility, fast iteration, and easier refinement across system versions. This architectural design not only supports the research goals of transparency, user control, and automation reliability, but also ensures the system remains extensible for future use cases and tools.

3.3.1 Version 1 - Script-Based Automation Framework

The first version of the system was developed to validate the core concept of **DevOps** automation triggered by natural language input. The primary goal was to build a functional end-to-end framework that could map user instructions to executable automation tasks using a fixed script-based architecture. This version served as a proof of concept and laid the technical foundation for more intelligent behaviour in later iterations.

The automation engine was implemented using a set of Python scripts, each corresponding to a specific **DevOps** scenario. These included scripts for generating GitHub Actions workflows, creating and building Docker containers, and simulating cloud deployment with tools like Pulumi. Example scripts included:

- `setup_github_actions.py` – Generates a **CI/CD** workflow in **YAML** format and commits it to the project repository.

- `dockerize_app.py` – Creates a Dockerfile, builds the image, and optionally runs the container.
- `setup_gitlab_ci.py` – Generates a `.gitlab-ci.yml` pipeline configuration for GitLab-based **CI/CD** automation.
- `setup_kubernetes.py` – Creates Kubernetes deployment and service configuration files and adds them to the repository.
- `deploy_to_cloud.py` – Generates infrastructure-as-code files (e.g., using Pulumi) to provision cloud resources for deployment.

The system architecture followed a simple client-server model. The frontend, built in React, provided a minimal **UI** with a text input field and a basic log display. The backend, implemented using FastAPI, exposed Representational State Transfer (REST) **API** endpoints that directly mapped predefined user inputs to the corresponding Python scripts. Each input triggered a script execution, and the results were returned to the frontend to inform the user.

Interaction Flow

The interaction flow was linear and predefined. The system does not perform any reasoning, decision-making, or multi-step planning. In the automation part of the flowchart, events placed on the left side correspond only to GitHub Actions pipeline automation, while events on the right side apply only to the application containerisation scenario using Docker. Events positioned in the middle represent shared steps that are relevant for both automation scenarios. The following flowchart describes this interaction in detail, showing how the frontend, backend, and automation scripts collaborate to complete a task:

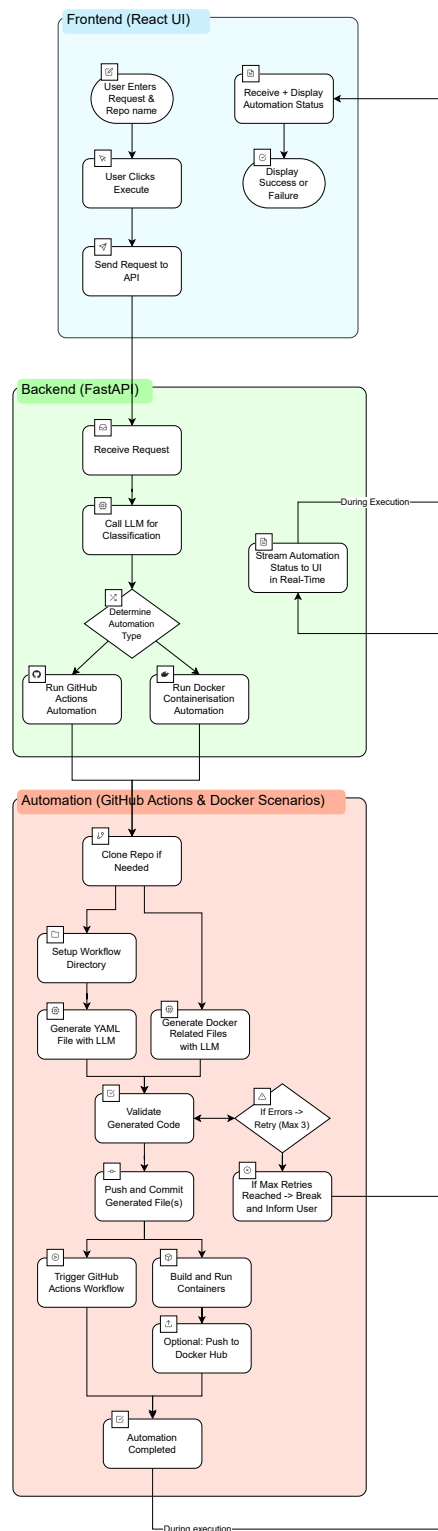


Figure 3.2: Interaction flow in the script-based system.

Frontend (React **UI)**

1. User Enters Request & Repo Name

The user writes a natural language prompt (e.g., "Create a GitHub Actions workflow for my React app") and enters their GitHub repository name.

2. User Clicks Execute

The user initiates the process by clicking the "Execute" button, which sends a request to the backend.

3. Send Request to **API**

A POST request is made to the FastAPI backend containing the prompt and repository name.

4. Receive + Display Final Output

After the script execution completes, the result (logs or summary) is returned and displayed in the **UI**.

Backend (FastAPI)

1. Receive Request

The backend receives and parses the incoming request.

2. Call **LLM for Classification**

Based on the user's prompt, the **LLM** maps the request to a specific pre-written Python script.

3. Run Automation Script

The backend executes the corresponding script using a subprocess.

4. Send Logs/Final Output to Frontend

After execution, logs or a final message are sent back to the frontend.

Automation Script (e.g., GitHub Actions, Docker)

1. Receive Repo Info

The script accepts the repository name and performs validation if necessary.

2. Clone Repository

The script clones the repository locally to allow for file generation and commits.

3. Generate Files

Based on templates or predefined logic, the script generates necessary files (e.g., **YAML** workflows or Dockerfiles).

4. Commit + Push Changes

The generated files are committed and pushed to the repository.

5. Return Success Log

A success message is returned, indicating the task completion.

This version had several notable strengths. It offered high accuracy in executing supported tasks and required no **LLM** inference, making it fully autonomous within its limited scope. However, the design had significant limitations. All logic was hardcoded, and the system lacked any form of **AI** reasoning or flexibility. It could not adapt to new input patterns, execute workflows that were not predefined, handle user errors, or dynamically modify execution plans. These constraints motivated the transition to a more flexible, agent-based design in the next system version.

3.3.2 Version 2 – Single-Agent AI DevOps Engineer

The second version of the system introduced a major architectural change by replacing hardcoded logic with a **ReAct**-style reasoning loop and human-in-the-loop control powered by a reasoning **LLM**-based **AI** agent. The goal of this version was to go beyond static script-based automation and allow the system to reason about user intent, dynamically plan actions, and involve the user in the decision-making process. This made the system more adaptable to a wider range of **DevOps** tasks and use cases.

Motivation

The primary motivation for this version was to overcome the limitations of Version 1. While the first version could execute predefined tasks accurately, it lacked flexibility and could not interpret variations in input or adapt to tasks beyond those explicitly scripted. By introducing **ReAct**-style reasoning (Thought → Action → Result), the system became capable of processing

natural language requests, proposing executable commands, and responding to feedback within an interactive loop.

Backend Modifications

The backend was extended to support a single **AI** agent based on an **LLM** (Claude Sonnet 3.5, before Qwen 2.5 coder and DeepSeek Coder v2 via Ollama). The agent follows a **ReAct**-style reasoning loop in which each cycle includes:

- **Thought:** Thinking about what needs to be done next based on the user's request and the current system state, aiming to find the most effective next step.
- **Action:** Suggesting a shell command or file generation task (e.g., `git clone, docker build, echo > Dockerfile`).
- **Result (Observation):** Capturing the execution result and feeding it back into the reasoning loop.

The system maintains simple loop memory, allowing the agent to reflect on previous results while reasoning through subsequent steps. The loop continues until the agent determines that the task is completed.

Frontend Enhancements

To support this reasoning loop, the frontend was updated to enable user-in-the-loop control. For each proposed action, the **UI** displays the agent's "Thought" and the corresponding command. Users can then choose to approve, edit, or reject the command. Approved commands are executed and the results are streamed back to the **UI** in real time. This interaction design promotes transparency, enhances safety, and builds trust in the **AI** system.

Interaction Flow

The flow for this version integrates frontend, backend, and **AI** reasoning components in a continuous cycle:

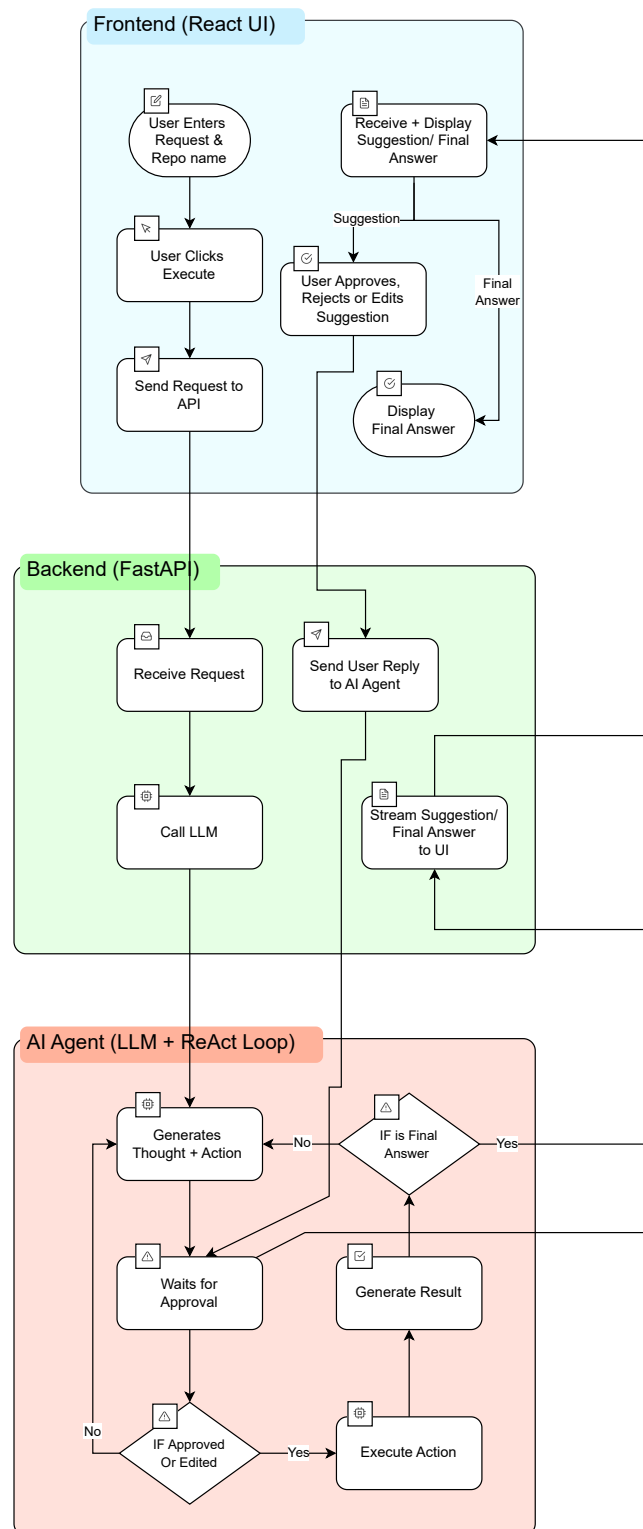


Figure 3.3: Interaction flow in the single-agent system.

Description of each step

Frontend (React **UI**)

1. **User Enters Request & Repo Name**
The user types a natural language request (e.g., "Create a GitHub Actions pipeline") and enters the repository name.
2. **User Clicks Execute**
Clicking the "Execute" button sends the input to the backend **API** to initiate the automation process.
3. **Send Request to **API****
A POST request is made to the FastAPI backend, triggering the automation process.
4. **Receive + Display System Logs/Answers**
The frontend listens for streamed logs and displays each step (Thought → Action → Result) as the **AI** agent reasons through the task (see Figure 3.4).
5. **User Approves, Rejects or Edits Suggestion**
When the **AI** suggests an action, the **UI** shows an editable command and buttons to approve, edit or reject it (see Figure 3.6).
6. **Display Final Answer**
Once the automation is complete, the final result or summary of what was done is displayed to the user.

Backend (FastAPI)

1. **Receive Request**
The backend receives the user request and repo name from the frontend.
2. **Call **LLM****
The backend uses an **LLM** (Claude Sonnet 3.5 in current version, before Qwen 2.5 coder and DeepSeek Coder v2 via Ollama) to interpret the request and generate the next action.
3. **Send User Reply to **AI** Agent**
When the user approves, edits, or rejects the suggestion, the backend passes this decision back to the agent to continue or stop.

4. Stream Suggestion/Final Answer to UI

All logs from the **AI** loop are streamed in real-time to the frontend for live feedback.

AI Agent (LLM + ReAct Loop)**1. Generates Thought + Action**

The agent uses an **LLM** to reason about the next step. It outputs a “Thought” (what needs to be done) and an “Action” (a shell command or file creation task) (see Figure 3.4).

2. Waits for Approval

The agent pauses and sends the proposed command to the frontend for user approval or editing (see Figure 3.5).

3. Execute Action (if approved)

Once approved, the backend executes the command and collects the result.

4. Generate Result

The outcome of the command is sent back to the **LLM** loop and displayed in the **UI**.

5. If Final Answer → Stop

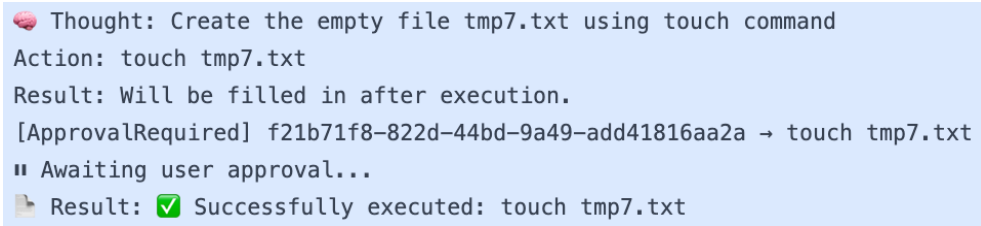
If the agent decides the task is complete, it sends a Final Answer and ends the loop.

6. Else → Repeat Loop

If not finished, the loop continues with a new Thought + Action.

Strengths and Limitations

This version marked a significant step forward in functionality and flexibility. It supported a wider range of **DevOps** tasks, leveraged transparent reasoning, and introduced interactive control for safer automation. However, it also had limitations in comparison to version 3. As a single agent system, it was limited in fallback and error handling, and its success heavily depended on the reliability of the output of a single **LLM** based **AI** agent. These challenges motivated the design of a multi-agent system in the next version.

A screenshot of a light blue rectangular box containing text that represents an AI agent's reasoning process. The text is organized into several lines, each starting with a small icon: a brain for 'Thought', a terminal icon for 'Action', a document for 'Result', a key for '[ApprovalRequired]', a notepad for 'Awaiting user approval...', and a folder for the final 'Result'.

🧠 Thought: Create the empty file tmp7.txt using touch command
💻 Action: touch tmp7.txt
📄 Result: Will be filled in after execution.
🔑 [ApprovalRequired] f21b71f8-822d-44bd-9a49-add41816aa2a → touch tmp7.txt
📝 " Awaiting user approval...
📁 Result: ✅ Successfully executed: touch tmp7.txt

Figure 3.4: The frontend displaying the AI agent's reasoning during execution, following the Thought → Action → Result loop.

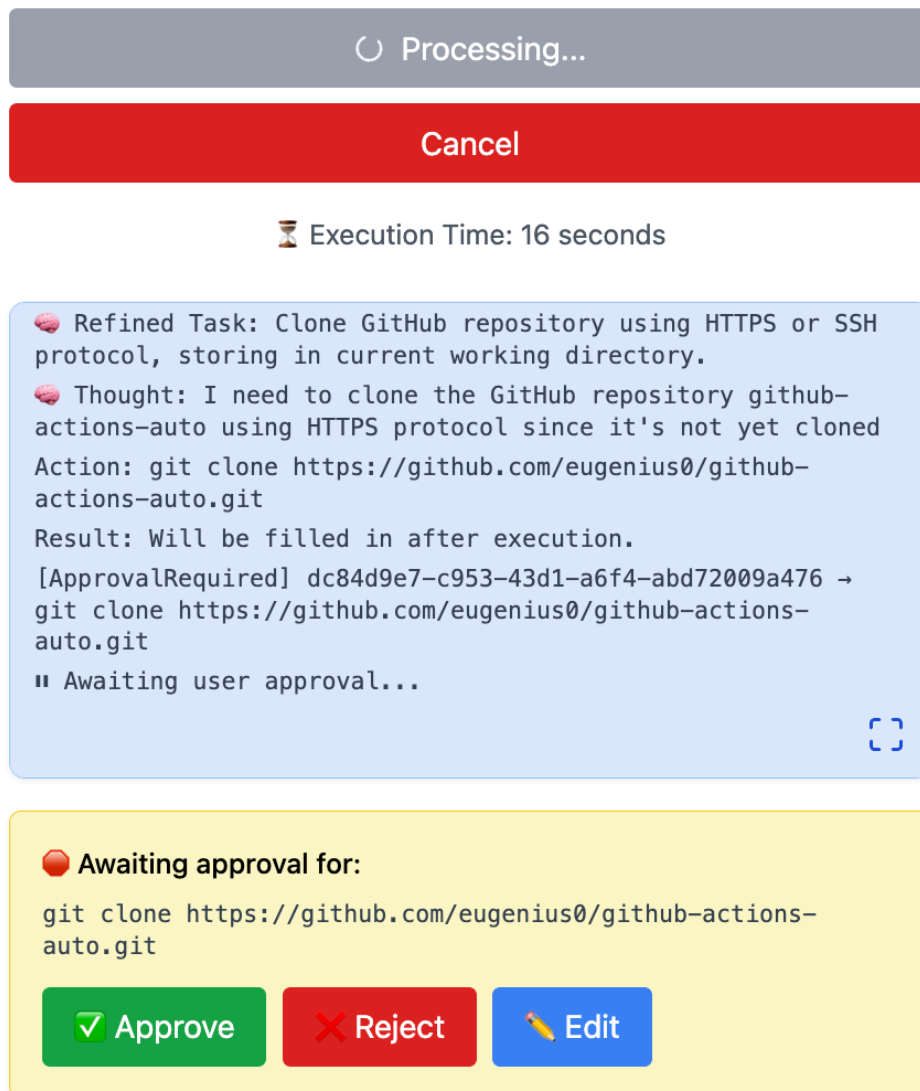


Figure 3.5: UI showing human-in-the-loop control in action as the system is waiting for user approval. Users can approve, edit, or reject the suggested command.

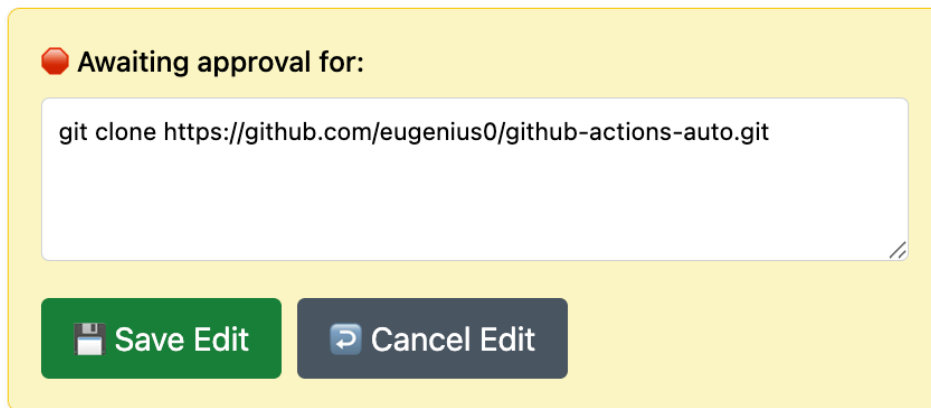


Figure 3.6: Edit mode allows users to adjust AI-generated commands before execution.

3.3.3 Version 3 – Multi-Agent AI DevOps Engineer

The third and final version of the system introduces a multi-agent architecture, designed to improve flexibility, error handling, and reasoning clarity beyond what was achievable in the single-agent model. By decomposing responsibilities into distinct agents, this version enables targeted behaviour, such as reflecting on rejected actions and improved reasoning. The modular structure also improves maintainability and sets the foundation for future extensibility.

Motivation

While the single-agent system brought significant improvements in automation and reasoning, it remained constrained by its monolithic structure. Handling rejected actions or rethinking flawed commands on the fly became increasingly complex to manage within a single reasoning loop. A modular agent-based architecture was therefore introduced to facilitate specialized roles, fallback mechanisms, and more robust decision-making.

Agent Architecture

This version introduces a coordinated system of three agents:

- **Reasoning Agent:** Interprets user intent, reasons through task steps, and proposes high-level actions.

- **Prompt Agent:** Constructs and optimizes prompts tailored to the context and task at hand, helping guide the **LLM** more effectively.
- **Reflector Agent:** Responds to rejected actions by analyzing the failure and proposing alternatives, improving resilience and adaptability.

These agents are orchestrated by a central component, the Agent Orchestrator, which manages the **ReAct**-style reasoning loop. It determines the agent interaction sequence, maintains the system state, and ensures smooth communication between agents. This coordination enables the system to handle more complex workflows, recover from errors, and provide more accurate results.

Unlike the previous version, the **ReAct** loop here is distributed across multiple agents, with each agent contributing its specialized capability to each Thought → Action → Result cycle. This explicit separation of concerns leads to clearer reasoning paths and improved traceability.

Backend Adjustments

To support the new architecture, the backend was refactored to manage multiple agents through the Agent Orchestrator. The orchestrator controls the flow of reasoning, handles user approvals, and dynamically delegates tasks to the appropriate agent. Execution remains asynchronous and event-driven, preserving responsiveness while coordinating inter-agent communication.

Frontend Enhancements

The **UI** was further improved to support the multi-agent workflow. Enhancements include:

- A fullscreen view for improved visibility (see Figure 3.7).
- Framer Motion-based animations for smoother transitions between reasoning steps and user inputs.

The frontend continues to display the Thought → Action → Result loop to the user, now showing the more granular decision making enabled by the agent architecture. Accordingly, the approval process and command editing interaction follow the same user-in-the-loop design described in Section 3.5 and Figure 3.6, but are now driven by modular agents.

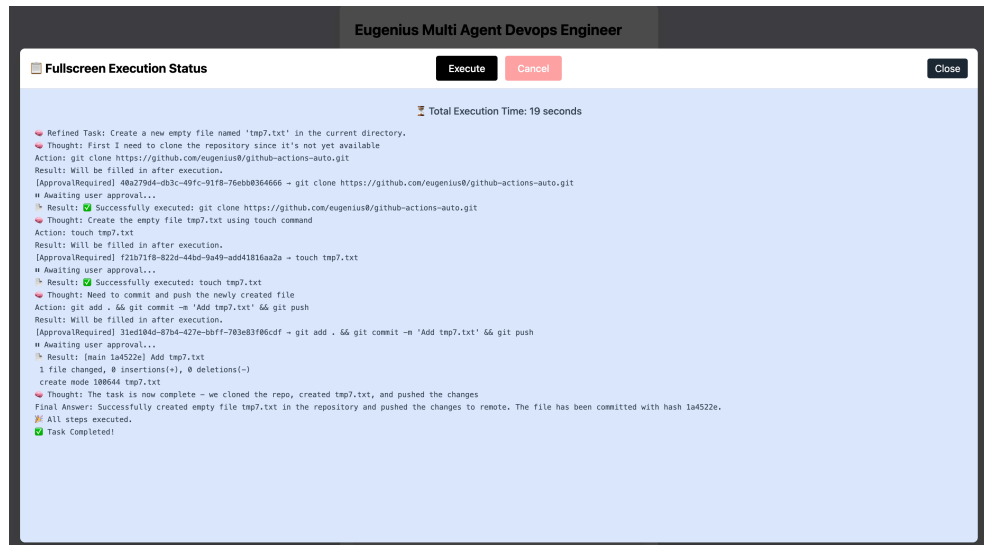


Figure 3.7: Fullscreen view of the **UI** in Version 3, showing the full Thought → Action → Result loop during task execution. The layout improves clarity and user experience.

Interaction Flow

The overall flow remains similar to Version 2 but is enhanced through modular agent participation which is shown in the following flowchart:

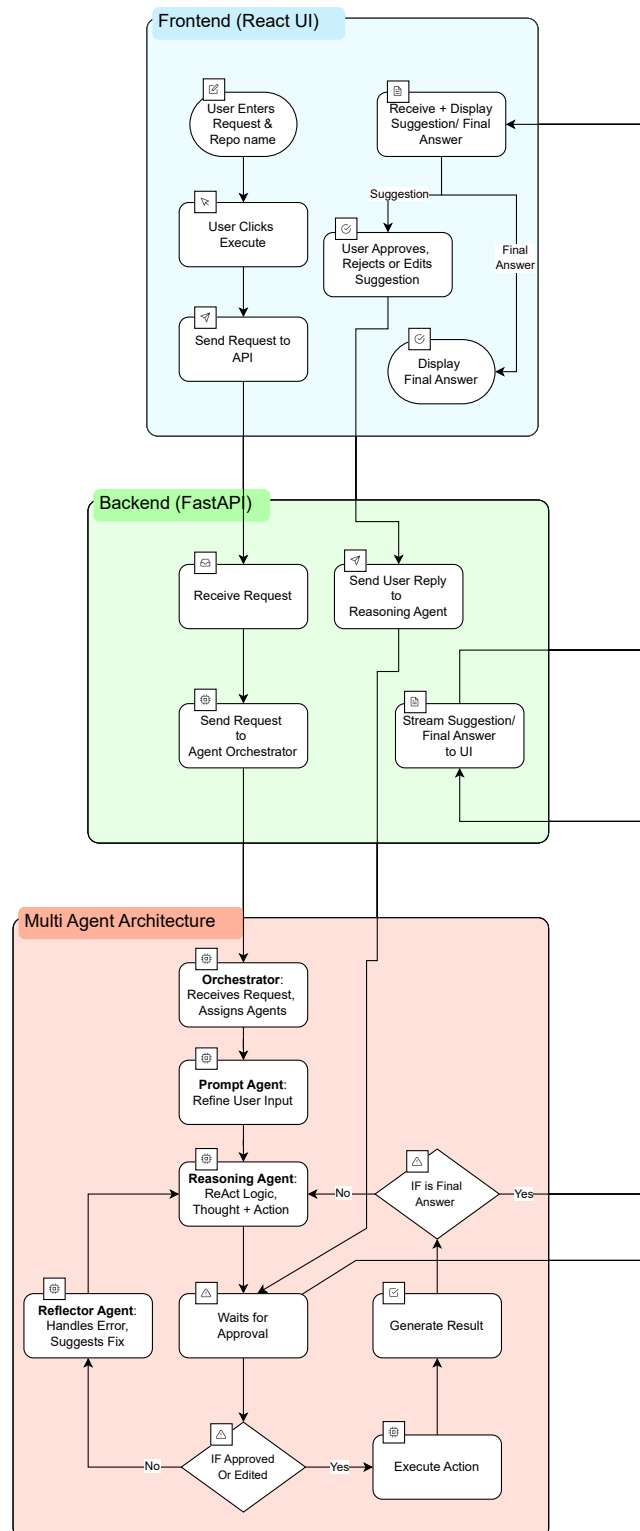


Figure 3.8: Interaction flow in the multi-agent system.

Description of each step:

Frontend (React UI)

1. **User Enters Request & Repo Name**
The user types a natural language request (e.g., "Create a GitHub Actions pipeline") and enters the repository name.
2. **User Clicks Execute**
Clicking the "Execute" button sends the input to the backend API to initiate the automation process.
3. **Send Request to API**
A POST request is made to the FastAPI backend, triggering the automation process.
4. **Receive + Display Suggestion/Final Answer**
The frontend listens for streamed logs and displays each step (Thought → Action → Result) as the AI agent reasons through the task.
5. **User Approves, Rejects or Edits Suggestion**
When the AI suggests an action, the UI shows a prompt with an editable command and buttons to approve, edit or reject it.
6. **Display Final Answer**
Once the automation is complete, the final result or summary of what was done is displayed to the user.

Backend (FastAPI)

1. **Receive Request**
The backend receives the user request and repo name from the frontend and passes it to the Agent Orchestrator.
2. **Send Request to Agent Orchestrator**
The orchestrator is responsible for coordinating all agents in the system.
3. **Send User Reply to Reasoning Agent**
When the user approves, edits, or rejects the suggestion, the backend passes this decision back to the reasoning agent via the Agent Orchestrator to continue or stop.

4. **Stream Suggestion/Final Answer to UI**

All logs from the Orchestrator and **AI** Agents are streamed in real-time to the frontend for live feedback.

Multi-Agent Architecture

1. **Agent Orchestrator: Receives Request, Assigns Agents**

Coordinates the task by invoking each specialised agent at the right moment.

2. **Prompt Engineer Agent: Refine User Input**

First, the prompt agent rewrites the user input into a precise automation goal.

3. **Reasoning Agent: ReAct Logic → Thought + Action**

Based on the refined input, the reasoning agent begins a **ReAct**-style loop, producing a Thought (reasoning) and a single Action (shell command).

4. **Waits for Approval**

The suggested command is paused and sent to the frontend for user confirmation.

5. **If Approved or Edited → Execute Action**

Once approved, the action is executed by the backend, and the output is returned to the agent.

6. **Generate Result**

The execution result is passed to the next loop iteration to inform future reasoning.

7. **If Result Fails → Reflector Agent**

If the command fails, the Reflector Agent analyses the error and suggests a fix, which is looped back into the agent's reasoning.

8. **If Final Answer → Stop**

When the Reasoning Agent declares the task complete, the system sends a Final Answer and ends the loop.

9. **Else → Repeat Loop**

If not finished, the loop continues with a new Thought → Action cycle.

Execution Workflow Description

This section describes the execution flow of the system for the task "*Create a GitHub Actions pipeline*", as visualized in Figure 3.9. Each stage highlights how the system components interact to interpret, execute, and respond to user intent.

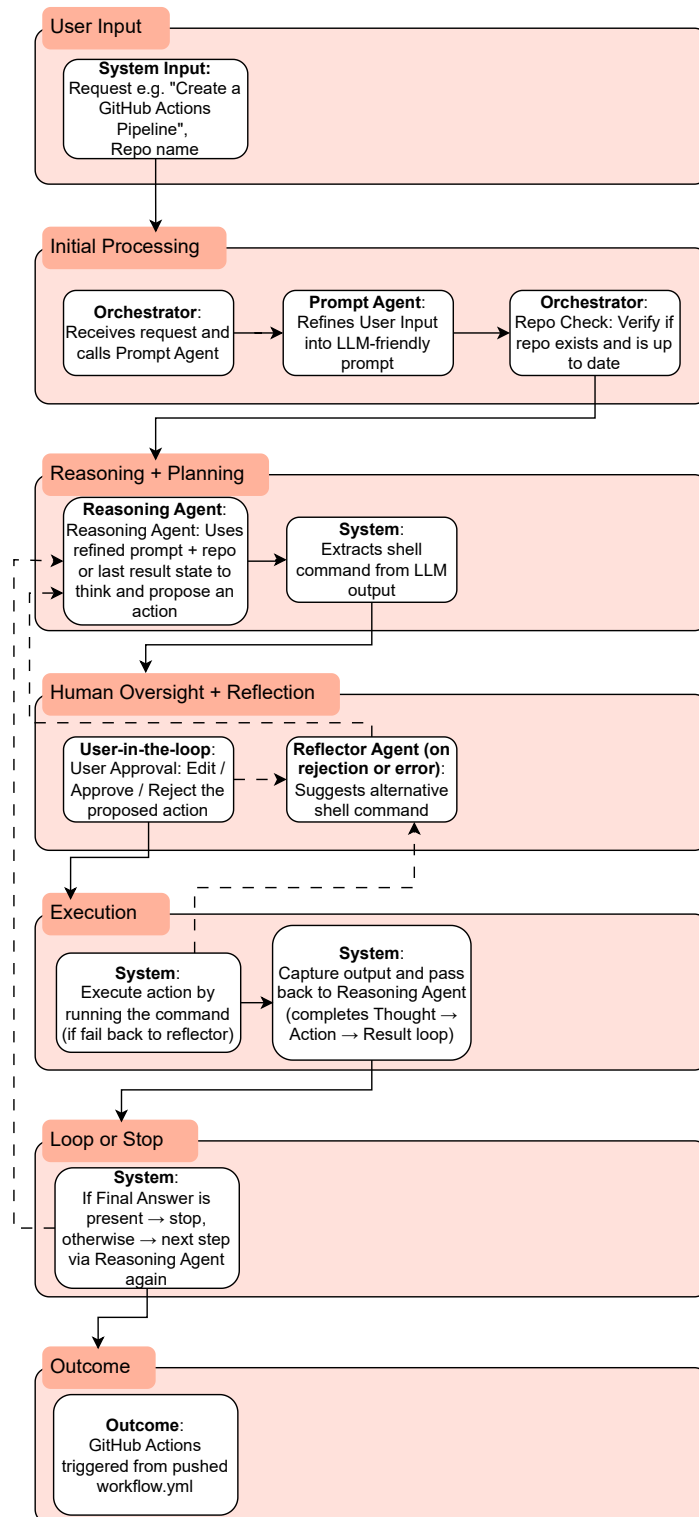


Figure 3.9: System execution workflow for the example task: "Create a GitHub Actions pipeline": Shows how the user input is processed and executed through coordinated agent interaction, resulting in the creation and deployment of `.github/workflows/workflow.yml`.

1. User Input

The user initiates the process by submitting a natural language request, such as *"Create a GitHub Actions pipeline"*, along with the repository name. This forms the starting point of the workflow.

2. Initial Processing

The Agent Orchestrator receives the request and passes it to the Prompt Agent, which rewrites the input into a clear and precise prompt suitable for the **LLM**. After prompt refinement, the orchestrator checks whether the repository exists locally and verifies if it is synchronized with the remote origin. This ensures that subsequent steps are based on the correct repository state.

3. Reasoning + Planning

The Reasoning Agent uses the refined input and current repo state or previous execution result or a recovery suggestion from the Reflector Agent to generate a reasoning step and a proposed shell command (the "Action"). This logic follows the Thought → Action → Result pattern. The system extracts the command from the **LLM** output in preparation for execution.

4. Human Oversight + Reflection

The proposed action is presented to the user for review. The user can approve, edit, or reject it. If the action is rejected, the Reflector Agent is invoked to suggest an alternative command to the Reasoning Agent. This ensures safe and transparent automation by keeping the user in control.

5. Execution

Once approved, the command is executed in the appropriate working directory. If the command fails during execution, the Reflector Agent is again activated to suggest a recovery command to the Reasoning Agent. This fallback mechanism allows the system to adapt dynamically to unexpected errors.

6. Result Loop

The result of the executed command is captured and passed back into the Reasoning Agent. This completes one iteration of the Thought → Action → Result loop. The system uses the output as context for the next reasoning step.

7. Loop or Stop

If the Reasoning Agent returns a *Final Answer*, the loop terminates. Otherwise, the system proceeds with another Thought → Action cycle using the updated context.

8. Outcome

Upon successful generation and execution of the command (e.g., creating and pushing `.github/workflows/workflow.yml`), the GitHub repository is updated. This triggers the GitHub Actions pipeline and marks the end of the automation workflow.

Strengths and Limitations

This version achieves a higher degree of modularity and accuracy, with improved error recovery and better adaptability to user feedback. The separation of responsibilities also makes the system easier to maintain and extend. However, these benefits come at the cost of increased system complexity and slower iteration cycles, due to the need for agent coordination and communication. Additionally, the system still lacks long-term learning or memory across sessions, and cannot yet self-improve based on experience.

3.4 Evaluation Approach

To evaluate the performance of the three developed system versions, script-based (v1), single-agent (v2), and multi-agent (v3), a structured test protocol was implemented. A total of 12 representative **DevOps** tasks were selected, covering a range of automation scenarios such as repository handling, **CI/CD** pipeline generation, containerisation, and simulated cloud deployment. Each task was executed three times per system version, resulting in 36 test runs per version, except for v1, which only supported 15 runs due to architectural limitations and lack of flexibility. The exact task descriptions and expected results are described in the following paragraph.

Evaluated Tasks

1. Clone a GitHub Repository

Input: "Clone my repo from GitHub"

Expected: Clones `https://github.com/<user>/test-app.git` into `./repos/test-app`

2. **Create a File**
Input: "Create the file tmp.txt"
Expected: Generates and commits a simple `tmp.txt`
3. **Delete a File**
Input: "Delete the file tmp.txt from my repo"
Expected: Removes the file, commits, and pushes the change
4. **Run `ls` or `git status`**
Input: "Show me the files in the repo"
Expected: Returns the output of `ls` or `git status`
5. **Generate a GitHub Actions Workflow**
Input: "Create a GitHub Actions pipeline for my app"
Expected: Generates a working `.github/workflows/ci.yml` file with valid YAML, commits, and pushes
6. **Containerize the App with Docker**
Input: "Containerise my app with Docker"
Expected: Generates a `Dockerfile` and optionally `docker-compose.yml`, commits, and pushes
7. **Start an already Dockerised App**
Input: "Start my already dockerised React app. Make sure the app is accessible on `http://localhost:3004`"
Expected: Runs `docker-compose up` or `docker run` and verifies the app is reachable at `localhost:3004`
8. **Create a GitLab Pipeline**
Input: "Create a GitLab pipeline for my app"
Expected: Generates a valid `.gitlab-ci.yml` file, commits, and pushes it to the repository
9. **Deploy to Kubernetes (Simulated)**
Input: "Deploy my app to Kubernetes with kubectl"
Expected: Generates `deployment.yml`, applies it using `kubectl` (simulated/test only)
10. **Recover from Repo Already Cloned**
Input: "Clone my repo from GitHub" (when it already exists)
Expected: Reflector Agent recognizes it's already cloned and continues in the correct directory

11. Simulated Merge Conflict

Input: User edits a file locally that conflicts with remote changes

Expected: System should recognize `git push` fails or recognize changes in the repository and resolve the conflict autonomously

12. Add Pulumi Deployment

Input: "Deploy my app to AWS with Pulumi"

Expected: Generates an initial Pulumi program (`index.ts`, `Pulumi.yaml`, `aws.ts`) and prepares deployment scripts (no actual deployment)

Test Execution Setup

All tasks were executed under controlled conditions: each system was assumed to start with no pre-cloned repositories, and user interactions were kept constant - **AI**-generated suggestions were always approved without manual modification. Moreover, the three systems were all based on the same **LLM** Claude Sonnet 3.5. This allowed for a consistent comparison across versions. Execution time was measured from task initiation to completion, including any delay introduced by user-in-the-loop approvals for v2 and v3. For each run, qualitative observations and quantitative metrics were recorded, including errors (during execution or in output), recovery behaviour, and success status.

Evaluation Dimensions

- **Functional Evaluation:** Measured the task success rate for each version, defined as tasks completed correctly without requiring human correction. This indicates the system's accuracy and output quality.
- **Efficiency Analysis:** Compared the average execution time per task across system versions. The analysis highlights differences in performance and responsiveness between versions.
- **Reliability Analysis:** Focused on the system's ability to recover from errors. This included logging how many errors occurred and how often the system recovered autonomously.

Considerations for Interpretation

Several contextual factors should be considered when analysing the results:

- **System Constraints:** Version 1 lacks flexibility and cannot attempt many tasks, which limits its comparability.
- **Task Clarity:** More concrete and specific task formulations were especially important for the LLM-powered versions (v2 and v3).
- **User-in-the-Loop Time Overhead:** For v2 and v3, efficiency results include the time needed for user approval, which could vary depending on reaction speed.
- **Development Investment:** Version 3 benefited from more time spent on refinement and prompt engineering, which may influence its superior results.
- **No User Correction:** In all tests, user suggestions were accepted as-is without adjustments; success rates could be higher with active user input.
- **Approval Consistency:** All actions suggested by the systems were approved, even if flawed, to ensure a neutral and reproducible evaluation baseline.

Data Recording and Visualization

Results were documented in a structured evaluation sheet which can be found in the appendix B and in the next chapter 4 visualised using a variety of diagrams.

User Experience

While not a primary focus of this thesis, user experience was evaluated through informal feedback from developers and test users. Feedback focused on the system's usability, transparency, and the clarity of the reasoning process as presented through the UI. This provided insights into how the Thought → Action → Result loop and the human-in-the-loop control affected user trust and interaction comfort.

3.5 Societal and Ethical Considerations

As intelligent automation systems increasingly influence software engineering practices, it becomes essential to consider their societal and ethical

implications. This thesis addresses several of these concerns through both its technical design and interaction model.

A key ethical principle guiding the system's development is transparency in **AI** decision-making. The use of a **ReAct**-style reasoning loop, structured as Thought → Action → Result, ensures that each decision made by the system is transparent and traceable. Rather than executing commands in a black-box fashion, the system explicitly reveals its reasoning process step by step, helping users understand why a particular action is suggested.

To reinforce user control and oversight, the system is designed with an interactive frontend, a so called user-in-the-loop control, that allows users to approve, edit, or reject each command before execution. This user-in-the-loop approach ensures that automation does not replace human judgment but instead augments it. By keeping users actively involved in the process, the system reduces the risk of blind trust in **AI**-generated decisions and at the same time increases the users trust in the system.

From a safety perspective, requiring user approval for all actions before execution minimises the potential for accidental or harmful operations. Whether caused by misinterpretation, ambiguous input, or model error, any potentially dangerous command can be intercepted and revised. This design choice enhances not only the technical robustness of the system but also its ethical alignment with principles of accountability and responsible **AI** use.

In addition, by lowering the technical barriers to complex DevOps automation, the system makes these capabilities accessible to a broader group of developers. This enables inclusivity and effective collaboration even for teams with developers that have varying levels of DevOps expertise.

Overall, the system provides a more transparent, controllable, and trustworthy form of **AI**-driven automation, which is essential when deploying such tools in sensitive, high-impact environments like **DevOps**.

Chapter 4

Results and Analysis

This chapter presents the results obtained from systematically evaluating the three system versions: the script-based automation framework (v1), the single-agent **AI DevOps** engineer (v2), and the multi-agent **AI DevOps** engineer (v3). The analysis is structured around three key dimensions: functionality, efficiency, reliability. Visualisations accompany each subsection to support the findings. All results are based on the structured testing protocol described in Chapter 3.4 which can be found in the appendix B.

In addition to the main evaluation dimensions, aspects of usability were also considered during development. Although not formally tested within the evaluation protocol, qualitative observations and interface design choices contributed to improving the system's **UX**.

4.1 Functional Evaluation

The functional evaluation focuses on measuring the task success rate of each system version. A task was considered successful if it was completed without requiring manual intervention or correction. This metric reflects each system's ability to interpret user input, generate appropriate automation steps, and execute them correctly.

As shown in Figure 4.1, the script-based version (v1) demonstrated a high success rate of 93.33%, completing 14 out of the 15 tasks it was capable of attempting. However, its limited flexibility meant it could not support the full set of 36 defined tasks. In contrast, the single-agent system (v2) attempted all tasks but succeeded on 25, yielding a success rate of 69.44

To further contextualise these results, Figure 4.2 compares the number of tasks each system attempted versus how many it successfully completed. This

distinction is especially important for v1, which could not attempt several tasks due to architectural constraints. The increased capabilities of v2 and v3 illustrate how LLM-based AI agent systems expand the range of supported automation scenarios, though success still varies with input clarity and internal reasoning performance.

Together, these results highlight clear improvements in both correctness and task coverage from the baseline script-based version to the AI-powered multi-agent architecture.

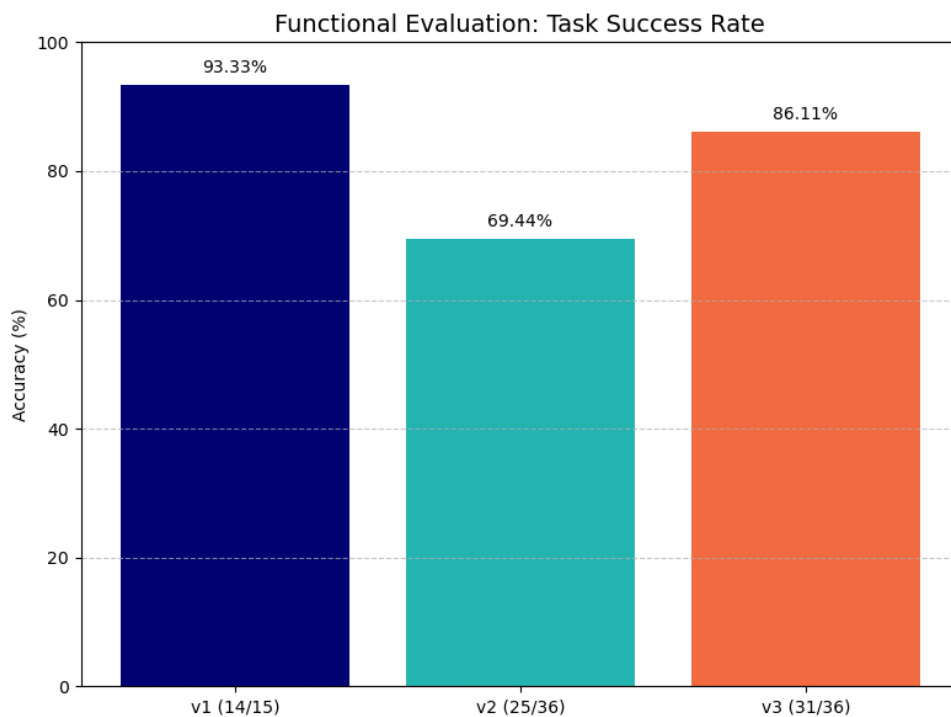


Figure 4.1: Task Success Rate per System Version: Percentage of tasks each system completed successfully without requiring manual correction.

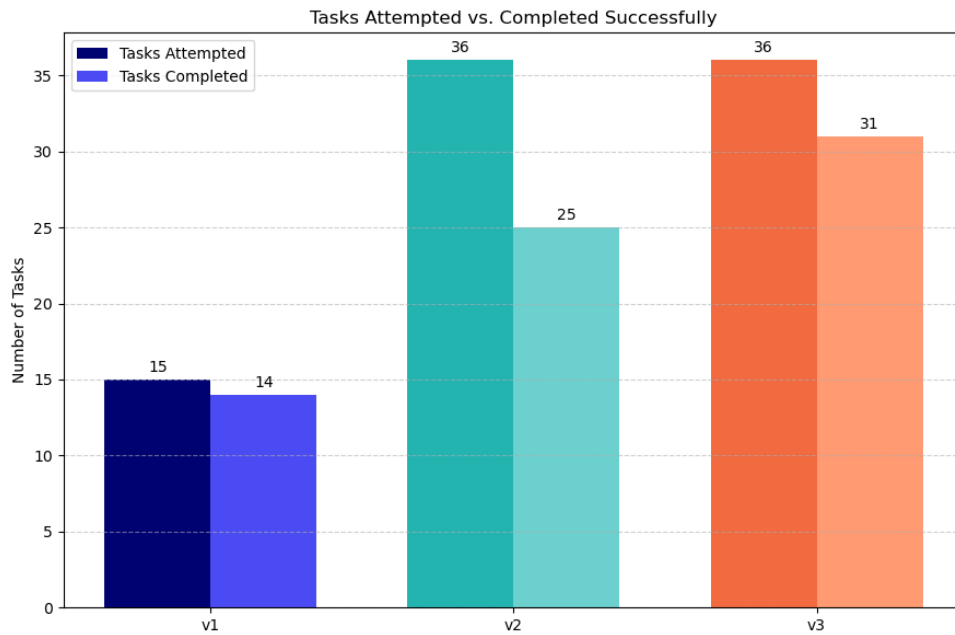


Figure 4.2: Tasks Attempted vs. Successfully Completed: Number of tasks each system attempted and how many were completed successfully.

4.2 Efficiency Analysis

Efficiency was assessed by measuring the average execution time per task for each system version. This includes the time taken to complete tasks end-to-end, encompassing user-in-the-loop approvals where applicable. The results, shown in Figure 4.3, indicate significant differences in execution times across the versions.

Version 1 (script-based) required the most time on average (75.80s per task), reflecting its limited automation and hard-coded structure. In contrast, both version 2 and version 3 achieved substantially lower execution times - 34.56s and 35.25s respectively - despite involving dynamic reasoning and user approvals. This demonstrates that the introduction of **AI**-driven automation not only adds flexibility but also reduces execution time.

It should be noted that for v2 and v3, user interaction (approval of **AI**-suggested commands) is part of the measured time. In real-world use, variations in user response time could influence these results further. Additionally, the baseline assumption for all systems was a clean execution environment with no pre-cloned repositories, ensuring consistency across trials.

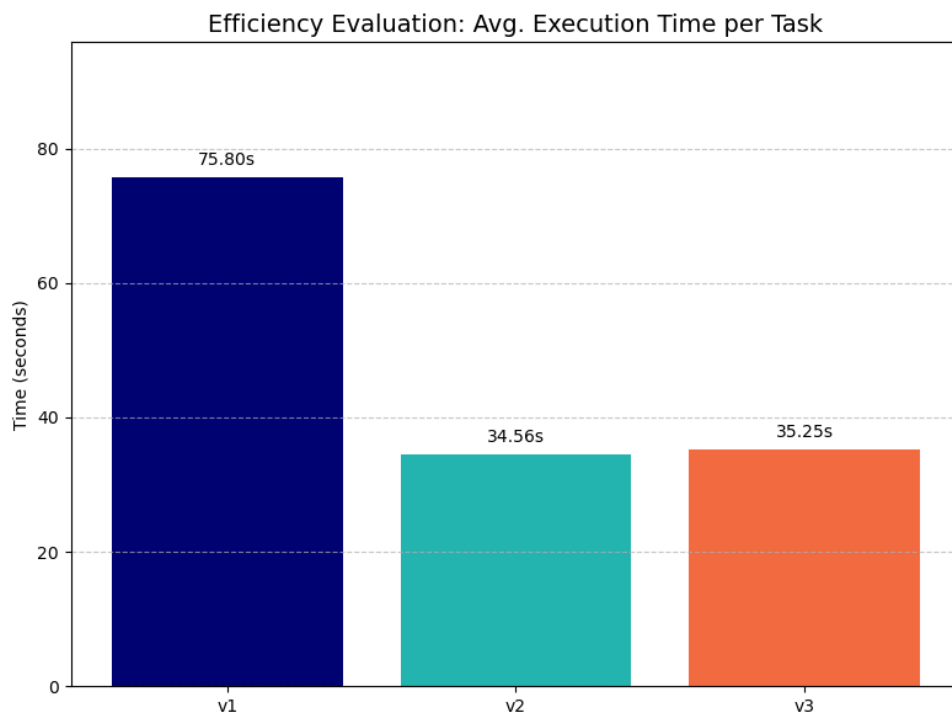


Figure 4.3: Average Execution Time per Task: Compares the average time taken to complete a task across the three system versions.

4.3 Reliability Analysis

Reliability was evaluated by examining each system's ability to recover from errors during execution. This includes errors arising from invalid shell commands, environment assumptions, or incomplete task handling. The metric of interest was the execution recovery rate, calculated as the ratio of successfully recovered execution errors to the total number of execution errors encountered.

As shown in Figure 4.4, version 1 encountered no execution errors during its limited test coverage, but this reflects its rigid nature rather than resilience. Version 2 experienced 36 execution errors and recovered from 28 of them, yielding a recovery rate of 77.78%. This strong performance demonstrates the single-agent system's ability to reason about failures and retry with alternative commands.

Version 3 encountered fewer execution errors overall (2), recovering from one (50.00%). Although this recovery rate is lower than v2, the small number of total errors in v3 shows that its improved reasoning and coordination

mechanisms and the advanced prompt reduce the likelihood of failure in the first place.

These results highlight the trade-off between robustness and reliability mechanisms. Version 2 shows strength in fallback handling, while version 3 demonstrates improved initial accuracy and fewer failure cases overall.

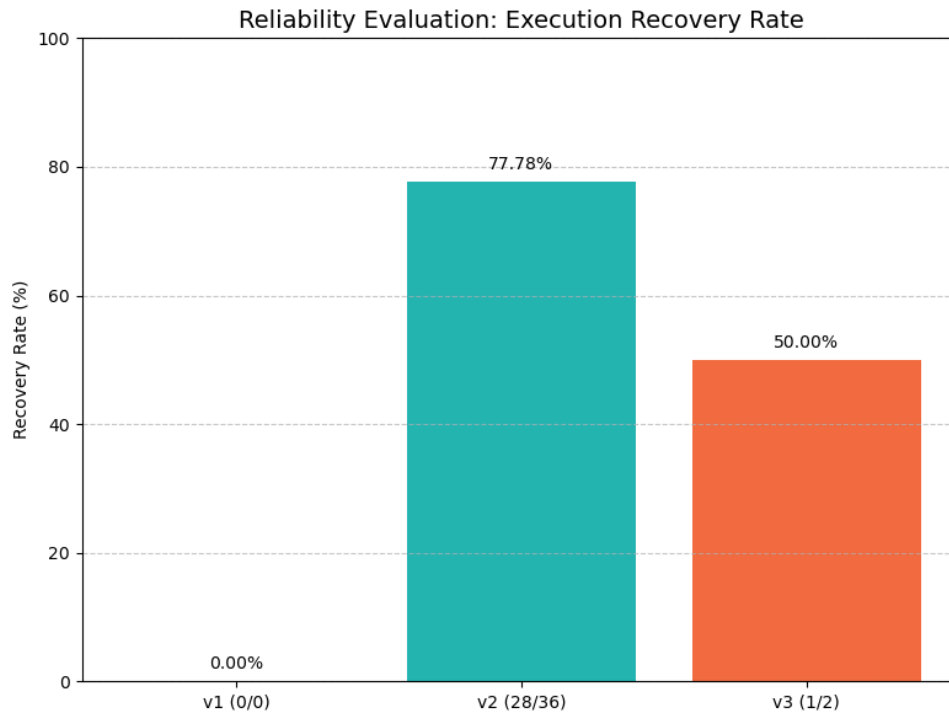


Figure 4.4: Execution Recovery Rate Across System Versions: Shows how reliably each system recovered from execution errors. **Note:** v3 had only 2 recovery opportunities, making its recovery rate less statistically representative than v2.

4.4 User Experience

Although user testing was not conducted as part of the evaluation, the systems were designed with user interaction and control in mind, particularly in versions 2 and 3. These versions incorporated a user-in-the-loop approval mechanism that allowed users to review, edit, or reject AI-generated shell commands before execution. Qualitative observations during testing revealed notable differences in perceived usability and interaction flow:

Version 1 required no user input during execution, but its rigid, script-based architecture offered no adaptability or transparency. There is no existing error recovery besides the hardcoded one, limiting its practicality in real-world usage.

Version 2 introduced a reasoning **AI** agent, **ReAct**-style reasoning and user approval interface. While **ReAct**-style reasoning improved transparency and the approval flow improved transparency and safety, the error recovery was limited as the system is dependent on one single agent so that manual intervention might be required by the user. However, this user-in-the-loop mechanism significantly increased user trust in the system, as users remained in control of each decision, enhancing security and preventing unintended commands from being executed.

Version 3 enhanced the **UX** by adopting a modular agent architecture, including a Reasoning Agent, Reflector Agent, and Prompt Agent. This made the system feel slower due to the additional communication between agents but also increased accuracy and error recovery. As a result, the system felt more intuitive for the user, with seamless transitions between alternative actions when initial suggestions were rejected.

In general, the Thought → Action → Result reasoning format helped users follow the system's logic and understand the rationale behind each decision. The multi-agent coordination in v3 added an additional layer of reliability and interpretability, supporting user trust through structured feedback loops.

4.5 Interpretation of Results

The evaluation of the three system versions - script-based (v1), single-agent (v2), and multi-agent (v3) - demonstrates an evolution in functionality, efficiency, reliability, and usability.

The functionality analysis shows that while version 1 achieved the highest success rate within its limited scope, this came at the cost of flexibility, as it was incapable of handling tasks beyond its hardcoded logic. Version 2 introduced **AI** agents and **AI** reasoning and was therefore capable of handling any kinds of tasks. Naturally it has a lower accuracy than a predefined script which is used in version 1 but therefore it is not limited to certain scenarios. Similar version 3 has the same flexibility. Its introduced modular agent architecture including different agents improved accuracy. It has to be noted though that more development time has been spent on version 3 than version 2. Version 3 has therefore for example a more detailed and improved prompt which has a high impact on the accuracy.

In the context of efficiency, the reduced execution times of versions 2 and 3 illustrate the time-saving potential of **AI**-driven automation. However, the role of user-in-the-loop control must be considered. While real-world deployments would involve varying user response times, the tests standardised this by assuming immediate approval of suggested commands by the user. This decision allowed the analysis to focus on inherent system performance rather than human factors. Interestingly, version 3's more complex architecture did not significantly increase execution time compared to version 2. That's related to the increased development time that has been spent on version 3 with advanced prompt engineering which decreased the amount of errors happening during execution. Additionally, version 3 introduced a prompt agent which craft and refines the prompt received by the user input for the system. Each error during execution highly increases the execution time as it requires a recovery from the system in our test case. This advantage of version 3 evens out with the disadvantage of it having slower iterations due to agent communication.

This leads us to the reliability analysis which showcases that version 2 had much more errors than version 3 during execution. As version 3 only had 2 errors during execution and could recover from 1 the presented recovery rate is not representative. Moreover, even without a reflector agent version 2 had a high recovery rate which illustrates the impact of introducing an **AI** agent for automation.

Finally, while usability was not a primary evaluation criterion and not directly measured through the testing protocol, it was considered during development. Improvements such as transparent reasoning steps, interactive command editing, and a fullscreen view for better visibility in version 3 contributed to a more transparent and accessible **UX**. These design choices support user trust and control, especially important in systems that involve user-in-the-loop decision-making.

In summary, version 3 emerged as the most robust and adaptable system, offering a compelling balance between automation, control, and error recovery. Version 2 represents a significant step forward from the rigidity of v1 but lacks the full coordination and fallback capabilities of v3.

Chapter 5

Conclusion and Future Work

5.1 Contributions

This thesis presents the design, development, and evaluation of three system versions for automating **DevOps** workflows: a script-based baseline (v1), a single-agent **AI DevOps** engineer (v2), and a multi-agent **AI DevOps** engineer (v3). The developed systems interpret natural language instructions and translate them into executable **DevOps** actions, covering tasks such as **CI/CD** pipeline setup, containerisation, and cloud deployment.

A key contribution is the introduction of **ReAct**-style reasoning, introduced in version 2, enabling the system to dynamically interpret user requests and generate automation steps with Thought → Action → Result loops. Version 3 further enhances this approach through modular agent coordination, involving dedicated Reasoning, Reflector, and Prompt agents. This architecture significantly improves flexibility, fallback handling, and transparency.

Another central contribution is the integration of user-in-the-loop control, which allows users to approve, reject, or edit each suggested action. This mechanism enhances safety and increases user trust in AI-driven automation by keeping users involved in critical decision points.

The goals of this work were met, as it has been demonstrated that automation workflows can be accelerated, accuracy can be improved, and user trust can be enhanced through transparent reasoning and user-in-the-loop control. Insights gained during development highlight the importance of agent modularity for fallback mechanisms, the critical role of transparency in building user trust in **AI**-generated decisions, and the necessity of user involvement to maintain control over automation steps.

For developers working on similar automation projects, the recommen-

dation is to start with a simple working baseline system. It is essential to first understand what exactly needs to be automated and why before scaling complexity. Building transparency into **AI** reasoning and maintaining user control over automation steps are crucial for ensuring safe and effective **AI** based systems.

In retrospect, an earlier focus on prompt engineering would have been beneficial. The impact of well-crafted prompts on system performance was underestimated in the initial stages, as well as the importance of selecting a capable **LLM** early in the process. Investing in a more advanced **LLM** from the beginning, even at higher cost, would have provided a clearer understanding of system limitations and guided more targeted improvements in prompt engineering in an earlier stage of the development process.

5.2 Future Work and Extensions

While the developed system demonstrates the feasibility of automating **DevOps** workflows using natural language and **AI** agents, several promising opportunities for future work remain.

A key aspect is extending support for additional **DevOps** tools and platforms beyond the ones tackled in this research such as Terraform or Jenkins. Testing the systems capability of working with other tools will expand the system's applicability and resilience in diverse environments.

Another enhancement involves equipping agents with persistent memory. This would enable the system to retain contextual knowledge across sessions, improving reasoning continuity, adaptability, and user personalisation. Persistent memory would allow agents to learn from past interactions, which optimises decision-making over time [83, 84, 85, 86].

Expanding the agent architecture with specialised roles, such as validator agents to ensure command safety, deployer agents for platform-specific operations, and coding agents specialised in generating high-quality code, would refine system functionality. In the future, a modular design where each tool (e.g., GitHub, GitLab, Docker) has a dedicated reasoning agent could further improve reasoning quality, system accuracy and task execution efficiency [87, 88]. This could even be enhanced by using a framework that depending on the task dynamically and autonomously creates specialised **AI** Agents [89].

Finally, more comprehensive testing, including edge cases and failure scenarios, is essential to validate robustness and fine-tune fallback mechanisms.

Simulating diverse environments, concurrent tasks, and abnormal conditions will strengthen the system's reliability and generalisability for real-world use.

These extensions will not only enhance system functionality but also reinforce the core principles of flexibility, transparency, and user control established in this work.

5.3 Limitations

The evaluation revealed several limitations in the current system that require consideration. First, the script-based version (v1) demonstrated high success rates but lacked flexibility, as it was incapable of handling tasks outside its predefined scope. This constraint limits its real-world applicability to specific scenarios.

In both the single-agent (v2) and multi-agent (v3) systems, the reliance on clear, concrete user requests highlighted the importance of prompt clarity and precision. Ambiguous or underspecified inputs led to suboptimal performance, suggesting that natural language input alone is not always sufficient for robust automation.

Execution efficiency measurements for v2 and v3 were influenced by the user-in-the-loop approval mechanism. Since approval speed can vary significantly between users and scenarios, the measured efficiency may not fully reflect real-world performance under variable user response times.

Another observed limitation concerns error handling. While v2 and v3 featured fallback mechanisms, v2's single-agent architecture limited recovery capabilities, and v3's recovery rate was impacted by increased complexity and slower iterations. Moreover, v3's modular architecture, while improving accuracy and flexibility, added additional communication overhead and introduced potential points of failure.

Additionally, all tests were conducted assuming a clean environment with no pre-cloned repositories. This setup does not fully capture the complexities of real-world development environments where partial states, pre-existing files, and concurrent changes might affect outcomes.

Finally, the system was tested with automatic user approval of suggestions, without simulating real-world user interventions or edits. This likely leads to not representative execution times and underestimates the potential of user corrections, especially in cases where skilled developers could adapt suggested commands more effectively.

5.4 Final Reflections

The development of the **AI**-powered **DevOps** automation system presented in this thesis offers valuable insights across economic, social, environmental, and ethical dimensions.

Economically, the system reduces the manual effort and time required to set up and manage **DevOps** workflows. This not only accelerates onboarding for new developers but also enables teams to focus more on value-generating tasks rather than repetitive configuration work.

Socially, the system makes complex **DevOps** automation accessible to a broader range of developers. By lowering the technical barrier, even those with limited **DevOps** experience can leverage automation to increase productivity and consistency, which enables inclusive collaboration within development teams.

From an environmental perspective, automation increases efficiency by reducing redundant processes and human errors that can result in wasted compute resources. Streamlined workflows contribute to lower energy consumption, especially in large-scale systems where inefficient configurations can lead to significant resource overhead.

Ethically, the system's design maintains human oversight as a central principle. By incorporating a user-in-the-loop mechanism and transparent **ReAct** reasoning, the system avoids the risks associated with blind automation. Users retain control over each decision, ensuring that the system's actions align with human judgment and ethical considerations.

In summary, the project demonstrates that **AI**-driven **DevOps** automation can achieve not only technical excellence but also meaningful contributions to efficiency, accessibility, environmental sustainability, and ethical responsibility.

In conclusion, this thesis demonstrates that combining Large Language Models with structured agent reasoning can significantly enhance the automation of **DevOps** workflows. By evaluating three distinct system versions, the project highlights the benefits of modular design, transparent **AI** decision-making, and user-in-the-loop control. The system not only achieves technical efficiency and robustness but also addresses economic, social, environmental, and ethical considerations, contributing to a future where **AI**-augmented **DevOps** is both powerful and responsible.

References

- [1] Amazon Web Services, “What is devops?” [Last updated: 2025-05-06. Accessed: 2025-06-09.]. [Online]. Available: <https://aws.amazon.com/devops/what-is-devops/> [Pages 1 and 7.]
- [2] Mike Tyson of the Cloud (MToC), “Devops workflow explained,” Dec. 2023, [Accessed: 2025-06-09.]. [Online]. Available: https://medium.com/@mike_tyson_cloud/devops-workflow-explained-6c2533c55d43 [Pages 1 and 7.]
- [3] Cortex.io, “Devops workflow: Overview, how-to, and tips,” Jul. 2024, [Accessed: 2025-06-09.]. [Online]. Available: <https://www.cortex.io/post/devops-workflows> [Page 1.]
- [4] V. U. Ugwueze and J. N. Chukwunweike, “Continuous integration and deployment strategies for streamlined devops in software engineering and application delivery,” *International Journal of Computer Applications Technology and Research*, vol. 14, no. 1, pp. 1–24, Jan. 2024. doi: 10.7753/IJCATR1401.1001 [Accessed: 2025-06-09.]. [Online]. Available: https://www.researchgate.net/publication/387601097_Continuous_Integration_and_Deployment_Strategies_for_Streamlined_DevOps_in_Software_Engineering_and_Application_Delivery [Page 1.]
- [5] M. H. Tanzil, M. Sarker, G. Uddin, and A. Iqbal, “A mixed method study of devops challenges,” Mar. 2024, [Accessed: 2025-06-09.]. [Online]. Available: <https://arxiv.org/pdf/2403.16436> [Pages 1, 3, 8, and 13.]
- [6] DuploCloud, “Devops implementation bottlenecks and how to overcome them,” Mar. 2025, accessed: 2025-06-09. [Online]. Available: <https://duplocloud.com/devops-implementation-bottlenecks/> [Pages 1, 3, and 8.]

- [7] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A comprehensive overview of large language models," Oct. 2024, [Accessed: 2025-06-09.]. [Online]. Available: <https://arxiv.org/pdf/2307.06435> [Page 1.]
- [8] A. Sobo, A. Mubarak, A. Baimagambetov, and N. Polatidis, "Evaluating llms for code generation in hri: A comparative study of chatgpt, gemini, and claude," *Applied Artificial Intelligence*, vol. 39, no. 1, p. 2439610, Dec. 2024. doi: 10.1080/08839514.2024.2439610 [Accessed: 2025-06-09.]. [Online]. Available: <https://www.tandfonline.com/doi/pdf/10.1080/08839514.2024.2439610> [Pages 2 and 9.]
- [9] M. Nejjar, L. Zacharias, F. Stiehle, and I. Weber, "Llms for science: Usage for code generation and data analysis," *Journal of Software: Evolution and Process*, Jul. 2024. doi: 10.1002/smr.2723 [Received: 2024-07-05; Revised: 2024-07-17; Accepted and Published: 2024-07-18. Accessed: 2025-06-09.]. [Online]. Available: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2723> [Pages 2 and 9.]
- [10] K. Valmeekam, M. Marquez, S. Sreedharan, and S. Kambhampati, "On the planning abilities of large language models: A critical investigation," Nov. 2023, [Accessed: 2025-06-09.]. [Online]. Available: <https://arxiv.org/abs/2305.15771> [Page 2.]
- [11] Y. Lu, S. Yang, C. Qian, G. Chen, Q. Luo, Y. Wu, H. Wang, X. Cong, Z. Zhang, Y. Lin, W. Liu, Y. Wang, Z. Liu, and M. Sun, "Proactive agent: Shifting llm agents from reactive responses to active assistance," Dec. 2024, [Accessed: 2025-06-09.]. [Online]. Available: <https://arxiv.org/pdf/2410.12361> [Page 2.]
- [12] Google Cloud, "What is an ai agent?" [Last updated: 2024-08-05. Accessed: 2025-06-09.]. [Online]. Available: <https://cloud.google.com/discover/what-are-ai-agents?hl=en> [Pages 2 and 11.]
- [13] M. A. Ferrag, N. Tihanyi, and M. Debbah, "From llm reasoning to autonomous ai agents: A comprehensive review," Apr. 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/pdf/2504.19678> [Pages 2 and 11.]
- [14] A. F. Khan, A. A. Khan, A. Mohamed, H. Ali, S. Moolinti, S. Haroon, U. Tahir, M. Fazzini, A. R. Butt, and A. Anwar, "Lads: Leveraging

- llms for ai-driven devops,” Feb. 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2502.20825> [Pages 2 and 12.]
- [15] Y. Cheng, C. Zhang, Z. Zhang, X. Meng, S. Hong, W. Li, Z. Wang, Z. Wang, F. Yin, J. Zhao, and X. He, “Exploring large language model based intelligent agents: Definitions, methods, and prospects,” Jan. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2401.03428> [Pages 2, 8, and 9.]
- [16] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, “React: Synergizing reasoning and acting in language models,” Mar. 2023, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/pdf/2210.03629> [Pages 2, 11, 12, and 14.]
- [17] G. He, G. Demartini, and U. Gadiraju, “Plan-then-execute: An empirical study of user trust and team performance when using llm agents as a daily assistant,” Feb. 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/pdf/2502.01390> [Page 2.]
- [18] A. Fourney, G. Bansal, H. Mozannar, C. Tan, E. Salinas, E. Zhu, F. Niedtner, G. Proebsting, G. Bassman, J. Gerrits, J. Alber, P. Chang, R. Loynd, R. West, V. Dibia, A. Awadallah, E. Kamar, R. Hosn, and S. Amershi, “Magentic-one: A generalist multi-agent system for solving complex tasks,” Nov. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/pdf/2411.04468v1> [Pages 2 and 13.]
- [19] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, “A design science research methodology for information systems research,” *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–78, Dec. 2007, accessed: 2025-06-09. [Online]. Available: <https://doi.org/10.2753/MIS0742-1222240302> [Page 4.]
- [20] J. vom Brocke, A. Hevner, and A. Maedche, “Introduction to design science research,” in *Design Science Research. Cases*, J. vom Brocke, A. Hevner, and A. Maedche, Eds. Springer, Cham, Sep. 2020, pp. 1–13, accessed: 2025-06-09. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-46781-4_1 [Page 4.]
- [21] K. Peffers, T. Tuunanen, C. E. Gengler, M. Rossi, W. Hui, V. Virtanen, and J. Bragge, “The design science research process: A model for producing and presenting information systems research,” Jun. 2020,

- accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/pdf/2006.02763> [Pages 4, 17, and 19.]
- [22] J. Salmons, “Case study methods and examples,” Feb. 2021, accessed: 2025-06-09. [Online]. Available: <https://researchmethodscommunity.sagepub.com/blog/case-study-methodology> [Page 4.]
- [23] J. de Lichtenberg, “The experimental approach,” Jan. 2017, accessed: 2025-06-09. [Online]. Available: <https://medium.com/the-experimental-approach/the-experimental-approach-e6a01a4b941e> [Page 4.]
- [24] Anthropic, “Claude 3.5 sonnet,” Jun. 2024, accessed: 2025-06-09. [Online]. Available: <https://www.anthropic.com/news/claude-3-5-sonnet> [Page 5.]
- [25] Ollama, “Get up and running with large language models.” 2025, accessed: 2025-06-09. [Online]. Available: <https://ollama.com> [Page 5.]
- [26] —, “deepseek-coder-v2,” 2024, [Last updated: 2024-09. Accessed: 2025-06-09]. [Online]. Available: <https://ollama.com/library/deepseek-coder-v2> [Page 5.]
- [27] —, “Qwen2.5-coder - model library entry,” 2024, [Last updated: 2025-06. Accessed: 2025-06-09]. [Online]. Available: <https://ollama.com/library/qwen2.5-coder> [Page 5.]
- [28] A. Crudu and M. R. Team, “The benefits of containerization - how docker and kubernetes revolutionize software development,” Mar. 2024, accessed: 2025-06-09. [Online]. Available: <https://moldstud.com/articles/p-the-benefits-of-containerization-with-docker-and-kubernetes> [Page 8.]
- [29] V. Krishnakumar, “Everything you need to know about containerization: Benefits, use cases, and best practices,” Jan. 2024, accessed: 2025-06-09. [Online]. Available: <https://www.cloudoptimo.com/blog/everything-you-need-to-know-about-containerization-benefits-use-cases-and-best-practices/> [Page 8.]
- [30] F. Paiva, J. Brunet, T. E. Pereira, and W. Oliveira, “A defect taxonomy for infrastructure as code: A replication study,” May 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2505.01568> [Page 8.]

- [31] A. Bordoloi, “Pulumi: A modern infrastructure as code (iac) alternative to terraform,” Jan. 2025, accessed: 2025-06-09. [Online]. Available: <https://devtron.ai/blog/introduction-to-pulumi-a-modern-infrastructure-as-code-platform/> [Page 8.]
- [32] H. da Gíão, A. Flores, R. Pereira, and J. Cunha, “Chronicles of ci/cd: A deep dive into its usage over time,” Feb. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2402.17588> [Page 8.]
- [33] T. A. Ghaleb, O. Abduljalil, and S. Hassan, “Ci/cd configuration practices in open-source android apps: An empirical study,” May 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2411.06077> [Pages 8 and 9.]
- [34] M. Wessel, T. Mens, A. Decan, and P. Rostami Mazrae, “The github development workflow automation ecosystems,” May 2023, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2305.04772> [Page 8.]
- [35] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, F. Luo, Y. Xiong, and W. Liang, “Deepseek-coder: When the large language model meets programming – the rise of code intelligence,” Jan. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2401.14196> [Page 9.]
- [36] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, K. Dang, Y. Fan, Y. Zhang, A. Yang, R. Men, F. Huang, B. Zheng, Y. Miao, S. Quan, Y. Feng, X. Ren, X. Ren, J. Zhou, and J. Lin, “Qwen2.5-coder technical report,” Nov. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2409.12186> [Page 9.]
- [37] C. Lyu, L. Yan, R. Xing, W. Li, Y. Samih, T. Ji, and L. Wang, “Large language models as code executors: An exploratory study,” Oct. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2410.06667> [Page 9.]
- [38] A. Haque, S. Siddique, M. M. Rahman, A. R. Hasan, L. R. Das, M. Kamal, T. Masura, and K. D. Gupta, “Sok: Exploring hallucinations and security risks in ai-assisted software development with insights for llm deployment,” Jan. 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2502.18468> [Page 9.]

- [39] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, and Y. Ma, “Exploring and evaluating hallucinations in llm-powered code generation,” May 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2404.00971> [Pages 9 and 10.]
- [40] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, “Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks,” Jul. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2312.12575> [Page 9.]
- [41] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, “A prompt pattern catalog to enhance prompt engineering with chatgpt,” Feb. 2023, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2302.11382> [Page 10.]
- [42] V. Gadesha, “What is prompt engineering?” Sep. 2025, accessed: 2025-06-09. [Online]. Available: <https://www.ibm.com/think/topics/prompt-engineering> [Page 10.]
- [43] D. Mehta, K. Rawool, S. Gujar, and B. Xu, “Automated devops pipeline generation for code repositories using large language models,” Dec. 2023, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2312.13225> [Page 10.]
- [44] H. Taherkhani, M. Sepidband, H. V. Pham, S. Wang, and H. Hemmati, “Epic: Cost-effective search-based prompt engineering of llms for code generation,” Aug. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2408.11198> [Page 10.]
- [45] M. Bruni, F. Gabrielli, M. Ghafari, and M. Kropp, “Benchmarking prompt engineering techniques for secure code generation with gpt models,” Feb. 2025, accepted at the 2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge 2025), Accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2502.06039> [Page 10.]
- [46] Y. Nasr, “Agentic ai in devops: The future of automation with a human touch,” Feb. 2025, accessed: 2025-06-09. [Online]. Available: <https://yanofnasr.medium.com/agentic-ai-in-devops-the-future-of-automation-with-a-human-touch-342e454ff5d7> [Pages 11 and 12.]

- [47] M. Renze and E. Guven, "Self-reflection in llm agents: Effects on problem-solving performance," Oct. 2024, accessed: 2025-06-18. [Online]. Available: <https://arxiv.org/abs/2405.06682> [Page 11.]
- [48] N. Krishnan, "Ai agents: Evolution, architecture, and real-world applications," Mar. 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2503.12687> [Page 11.]
- [49] H. Bhoite, "Autonomous ai agents for end-to-end data engineering pipelines deployment: Enhancing ci/cd pipelines," May 2025, accessed: 2025-06-09. [Online]. Available: https://d197for5662m48.cloudfront.net/documents/publicationstatus/257481/preprint_pdf/88a6adbd4853e5ad881d77efcd83f527.pdf [Page 11.]
- [50] A. Masood, "Autonomous ai agents and adjacent automation technologies," May 2025, accessed: 2025-06-09. [Online]. Available: <https://medium.com/@adnanmasood/autonomous-ai-agents-and-adjacent-automation-technologies-08f22bd8245b> [Page 12.]
- [51] R. Sapkota, K. I. Roumeliotis, and M. Karkee, "Ai agents vs. agentic ai: A conceptual taxonomy, applications and challenges," May 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2505.10468> [Page 12.]
- [52] M. Rawat, A. Gupta, R. Goomer, A. Di Bari, N. Gupta, and R. Pieraccini, "Pre-act: Multi-step planning and reasoning improves acting in llm agents," May 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2505.09970> [Page 12.]
- [53] S. Li, H. Xu, and H. Chen, "Focused react: Improving react through reiterate and early stop," Oct. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2410.10779> [Page 12.]
- [54] Z. Wu, "Autono: A react-based highly robust autonomous agent framework," Apr. 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2504.04650> [Page 12.]
- [55] A. Bilal, D. Ebert, and B. Lin, "Llms for explainable ai: A comprehensive survey," Mar. 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2504.00125> [Page 12.]
- [56] Asycd, "React prompt framework: Enhancing ai's decision making with human-like reasoning," Jun. 2024, accessed: 2025-06-09. [Online].

- Available: <https://asycd.medium.com/react-prompt-framework-enhancing-ais-decision-making-with-human-like-reasoning-72a30df34ead> [Page 12.]
- [57] C. Greyling, “Ai agents with human in the loop,” Aug. 2024, accessed: 2025-06-09. [Online]. Available: <https://cobusgreyling.medium.com/a-i-agents-with-human-in-the-loop-f910d0c0384b> [Pages 12 and 19.]
- [58] B. Lou, T. Lu, T. S. Raghu, and Y. Zhang, “Unraveling human-ai teaming: A review and outlook,” Apr. 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2504.05755> [Page 12.]
- [59] Q. Gao, W. Xu, H. Pan, M. Shen, and Z. Gao, “Human-centered human-ai collaboration (hchac),” May 2025, to appear in the Handbook of Human-Centered Artificial Intelligence, Accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2505.22477> [Pages 12 and 13.]
- [60] J. He, C. Treude, and D. Lo, “Llm-based multi-agent systems for software engineering: Literature review, vision and the road ahead,” Dec. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2404.04834> [Pages 13 and 14.]
- [61] Y. Aman, “Multi-agent orchestration overview,” Jan. 2025, accessed: 2025-06-09. [Online]. Available: <https://medium.com/@yugank.aman/multi-agent-orchestration-overview-aa7e27c4e99e> [Page 13.]
- [62] C. Bronsdon, “Centralized vs distributed multi-agent ai coordination strategies,” Apr. 2025, accessed: 2025-06-09. [Online]. Available: <https://galileo.ai/blog/multi-agent-coordination-strategies> [Page 13.]
- [63] K.-T. Tran, D. Dao, M.-D. Nguyen, Q.-V. Pham, B. O’Sullivan, and H. D. Nguyen, “Multi-agent collaboration mechanisms: A survey of llms,” Jan. 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2501.06322> [Page 13.]
- [64] H. van Ditmarsch, K. Fruzsa, R. Kuznets, and U. Schmid, “A logic for repair and state recovery in byzantine fault-tolerant multi-agent systems,” in *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2024)*, ser. Lecture Notes in Artificial Intelligence, vol. 14740. Springer, June 2024. doi: 10.1007/978-3-031-63501-4_7 pp. 114–134, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2401.06451> [Page 13.]

- [65] W. Jin, H. Du, B. Zhao, X. Tian, B. Shi, and G. Yang, “A comprehensive survey on multi-agent cooperative decision-making: Scenarios, approaches, challenges and perspectives,” Mar. 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2503.13415> [Page 13.]
- [66] J. Hu, Y. Dong, S. Ao, Z. Li, B. Wang, L. Singh, G. Cheng, S. D. Ramchurn, and X. Huang, “Position: Towards a responsible llm-empowered multi-agent systems,” Feb. 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2502.01714> [Page 13.]
- [67] S. G. Saroar and M. Nayebi, “Developers’ perception of github actions: A survey analysis,” Mar. 2023, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2303.04084> [Page 13.]
- [68] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, “Learning from, understanding, and supporting devops artifacts for docker,” in *Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020)*. ACM, May 2020. doi: 10.1145/3377811.3380406 Accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2002.03064> [Page 13.]
- [69] X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao, “Self-planning code generation with large language models,” *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 1, pp. 1–31, Oct. 2024. doi: 10.1145/XXXXXXX.XXXXXXX Accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2303.06689> [Page 14.]
- [70] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, “A survey on large language models for code generation,” Nov. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2406.00515> [Page 14.]
- [71] A. Almorsi, M. Ahmed, and W. Gomaa, “Guided code generation with llms: A multi-agent framework for complex code tasks,” in *Proceedings of the 2024 IEEE International Japan-Africa Conference on Electronics, Communications and Computations (JAC-ECC)*. IEEE, Jan. 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2501.06625> [Page 14.]

- [72] Y. Zhang, R. Sun, Y. Chen, T. Pfister, R. Zhang, and S. . Arik, “Chain of agents: Large language models collaborating on long-context tasks,” Jun. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2406.02818> [Page 14.]
- [73] OpenAI, “Introducing operator,” Jan. 2025, accessed: 2025-06-11. [Online]. Available: <https://openai.com/index/introducing-operator/> [Page 14.]
- [74] A. Duque, A. Syed, K. V. Day, M. J. Berry, D. S. Katz, and V. V. Kindratenko, “Leveraging large language models to build and execute computational workflows,” in *Proceedings of the 18th Workshop on Workflows in Support of Large-Scale Science (WORKS’23)*. ACM, Nov. 2023. doi: 10.1145/3624062.3624220 Accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2312.07711> [Page 14.]
- [75] W. Zhou, M. Mesgar, A. Friedrich, and H. Adel, “Efficient multi-agent collaboration with tool use for online planning in complex table question answering,” Feb. 2025, accepted at NAACL 2025 Findings, Accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2412.20145> [Page 14.]
- [76] Y. Ishibashi and Y. Nishimura, “Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization,” Apr. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2404.02183> [Page 14.]
- [77] GeeksforGeeks, “Client-server architecture – system design,” Jul. 2024, accessed: 2025-06-09. [Online]. Available: <https://www.geeksforgeeks.org/client-server-architecture-system-design/> [Page 19.]
- [78] Meta Platforms, Inc., “React – the library for web and native user interfaces,” 2025, accessed: 2025-06-09. [Online]. Available: <https://react.dev> [Page 19.]
- [79] S. Ramírez, “Fastapi,” 2025, accessed: 2025-06-09. [Online]. Available: <https://fastapi.tiangolo.com> [Page 19.]
- [80] Tailwind Labs, “Tailwind css - rapidly build modern websites without ever leaving your html,” 2025, accessed: 2025-06-09. [Online]. Available: <https://tailwindcss.com> [Page 20.]

- [81] —, “Headless ui – completely unstyled, fully accessible ui components, designed to integrate beautifully with tailwind css.” 2025, accessed: 2025-06-09. [Online]. Available: <https://headlessui.com> [Page 20.]
- [82] Framer, “Framer motion – a robust animation library for modern web projects using javascript, react, or vue.” 2025, accessed: 2025-06-09. [Online]. Available: <https://motion.dev> [Page 20.]
- [83] R. Westhäußer, F. Berenz, W. Minker, and S. Zepf, “Caim: Development and evaluation of a cognitive ai memory framework for long-term interaction with intelligent agents,” May 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2505.13044> [Page 56.]
- [84] Y. Hou, H. Tamoto, and H. Miyashita, ““my agent understands me better”: Integrating dynamic human-like memory recall and consolidation in llm-based agents,” in *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems (CHI EA '24)*. ACM, May 2024. doi: 10.1145/3544549.3585674 Accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2404.00573> [Page 56.]
- [85] J. Wei, X. Ying, T. Gao, F. Bao, F. Tao, and J. Shang, “Ai-native memory 2.0: Second me,” Mar. 2025, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2503.08102> [Page 56.]
- [86] C. Stryker, “What is ai agent memory?” accessed: 2025-06-09. [Online]. Available: <https://www.ibm.com/think/topics/ai-agent-memory> [Page 56.]
- [87] D. Arora, A. Sonwane, N. Wadhwa, A. Mehrotra, S. Utpala, R. Bairi, A. Kanade, and N. Natarajan, “Masai: Modular architecture for software-engineering ai agents,” Jun. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2406.11638> [Page 56.]
- [88] J. R. Harper, “Autogenesisagent: Self-generating multi-agent systems for complex tasks,” Apr. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2404.17017> [Page 56.]
- [89] G. Chen, S. Dong, Y. Shu, G. Zhang, J. Sesay, B. F. Karlsson, J. Fu, and Y. Shi, “Autoagents: A framework for automatic agent generation,” Apr. 2024, accessed: 2025-06-09. [Online]. Available: <https://arxiv.org/abs/2309.17288> [Page 56.]

Appendix A

GitHub Repositories

The following repositories contain the implementation of each system version developed in this thesis:

- **Version 1 – Script-Based System:**

<https://github.com/Eugenius0/automation-framework>

- **Version 2 – Single-Agent System:**

<https://github.com/Eugenius0/ai-devops-engineer>

- **Version 3 – Multi-Agent System:**

<https://github.com/Eugenius0/multi-agent-devops-engineer>

Appendix B

Test Evaluation Table

Task #	Task	System Version	Run	Functional (Y/N)	Time (sec)	Errors (If Any)	Recovered (Y/N)	Notes
1	Clone GitHub Repo	V1	1	N	-	-	-	Task not supported by v1
1	Clone GitHub Repo	V1	2	N	-	-	-	Task not supported by v1
1	Clone GitHub Repo	V1	3	N	-	-	-	Task not supported by v1
1	Clone GitHub Repo	V2	1	Y	-	-	-	-
1	Clone GitHub Repo	V2	2	Y	7	-	-	-
1	Clone GitHub Repo	V2	3	Y	6	-	-	-
1	Clone GitHub Repo	V2	3	Y	6	-	-	-
1	Clone GitHub Repo	V3	1	Y	7	-	-	-
1	Clone GitHub Repo	V3	2	Y	16	-	-	-
1	Clone GitHub Repo	V3	3	Y	11	-	-	-
2	Create tmp.txt File	V1	1	N	-	-	-	Task not supported by v1
2	Create tmp.txt File	V1	2	N	-	-	-	Task not supported by v1
2	Create tmp.txt File	V1	3	N	-	-	-	Task not supported by v1
2	Create tmp.txt File	V2	1	Y	16	-	-	-
2	Create tmp.txt File	V2	2	Y	33	-	-	-
2	Create tmp.txt File	V2	3	Y	26	-	-	-
2	Create tmp.txt File	V3	1	Y	12	-	-	-
2	Create tmp.txt File	V3	2	Y	25	-	-	-
2	Create tmp.txt File	V3	3	Y	16	-	-	-
3	Delete tmp.txt File	V1	1	N	-	-	-	Task not supported by v1
3	Delete tmp.txt File	V1	2	N	-	-	-	Task not supported by v1
3	Delete tmp.txt File	V1	3	N	-	-	-	Task not supported by v1
3	Delete tmp.txt File	V2	1	Y	16	-	-	-
3	Delete tmp.txt File	V2	2	Y	14	-	-	-
3	Delete tmp.txt File	V2	3	Y	15	-	-	-
3	Delete tmp.txt File	V3	1	Y	16	-	-	-
3	Delete tmp.txt File	V3	2	Y	15	-	-	-
3	Delete tmp.txt File	V3	3	Y	14	-	-	-

Figure B.1: Test Evaluation Table - Part 1: Tasks 1–3

4	Run ls or git status	V1	1	N	-	-	-	Task not supported by v1
4	Run ls or git status	V1	2	N	-	-	-	Task not supported by v1
4	Run ls or git status	V1	3	N	-	-	-	Task not supported by v1
4	Run ls or git status	V2	1	Y	18	No such file or directory	Y	Common error where it forgets that it is already in the right directory
4	Run ls or git status	V2	2	Y	17	Same error again	Y	-
4	Run ls or git status	V2	3	Y	17	Same error again	Y	-
4	Run ls or git status	V3	1	Y	10	-	-	-
4	Run ls or git status	V3	2	Y	9	-	-	-
4	Run ls or git status	V3	3	Y	11	-	-	-
5	Generate GitHub Actions Workflow	V1	1	N	57	Error in code	N	Error in generated yaml file
5	Generate GitHub Actions Workflow	V1	2	Y	52	-	-	-
5	Generate GitHub Actions Workflow	V1	3	Y	47	-	-	-
5	Generate GitHub Actions Workflow	V2	1	N	30	Error in code	N	Error in generated yaml file
5	Generate GitHub Actions Workflow	V2	2	N	45	Error in code	N	Error in generated yaml file
5	Generate GitHub Actions Workflow	V2	3	N	42	Error in execution	N	Generated code but didn't add it to the new created file
5	Generate GitHub Actions Workflow	V3	1	N	78	Error in execution + code, unexpected EOF	Y	Adjusted now reasoning agent prompt file I did before for v2
5	Generate GitHub Actions Workflow	V3	2	N	26	Error in code	N	Fixed the adding code to command suggestion problem
5	Generate GitHub Actions Workflow	V3	3	N	28	Error in code	N	Generated GitHub Actions Pipeline sometimes still has some issues but faster execution after fix
6	Containerize App with Docker	V1	1	Y	35	-	-	-
6	Containerize App with Docker	V1	2	Y	37	-	-	-
6	Containerize App with Docker	V1	3	Y	27	-	-	-
6	Containerize App with Docker	V2	1	N	20	Error in execution	N	Similar to error in task 5 in generated the code and file but didnt add code
6	Containerize App with Docker	V2	2	N	24	Error in execution	N	Similar to error in task 5 in generated the code and file but didnt add code
6	Containerize App with Docker	V2	3	N	23	Error in execution	N	Similar to error in task 5 in generated the code and file but didnt add code
6	Containerize App with Docker	V3	1	Y	158	-	-	App was successfully available under localhost, took long because of docker build -t react-docker:app .
6	Containerize App with Docker	V3	2	Y	62	-	-	-
6	Containerize App with Docker	V3	3	Y	83	-	-	-

Figure B.2: Test Evaluation Table - Part 2: Tasks 4–6

74 | Appendix B: Test Evaluation Table

7	Start Dockerised App (http://localhost:3004)	V1	1	N	-	-	-	-	Task not supported by v1
7	Start Dockerised App (http://localhost:3004)	V1	2	N	-	-	-	-	Task not supported by v1
7	Start Dockerised App (http://localhost:3004)	V1	3	N	-	-	-	-	Task not supported by v1
7	Start Dockerised App (http://localhost:3004)	V2	1	Y	-	26	-	-	Gets stuck in user in control loop but did the task successfully
7	Start Dockerised App (http://localhost:3004)	V2	2	Y	-	31	-	-	This time worked successfully
7	Start Dockerised App (http://localhost:3004)	V2	3	Y	-	38	-	-	Gets stuck in user in control loop but did the task successfully
7	Start Dockerised App (http://localhost:3004)	V3	1	Y	-	30	-	-	-
7	Start Dockerised App (http://localhost:3004)	V3	2	Y	-	28	-	-	-
7	Start Dockerised App (http://localhost:3004)	V3	3	Y	-	41	-	-	Took longer because system wanted to check the http response to verify accessibility
8	Create GitLab Pipeline	V1	1	Y	-	27	-	-	-
8	Create GitLab Pipeline	V1	2	Y	-	18	-	-	-
8	Create GitLab Pipeline	V1	3	Y	-	21	-	-	-
8	Create GitLab Pipeline	V2	1	N	-	23	Error in execution	N	System got confused by my way of providing my repo name containing project and repo name
8	Create GitLab Pipeline	V2	2	N	-	17	Error in execution	N	Same error (high potential to get fixed with some adjustments of the system for gitlab)
8	Create GitLab Pipeline	V2	3	N	-	21	Error in execution	N	Same error (high potential to get fixed with some adjustments of the system for gitlab)
8	Create GitLab Pipeline	V3	1	Y	-	38	-	-	Prompt fix in v3 to make it work but rather a temporary solution for test purposes
8	Create GitLab Pipeline	V3	2	N	-	34	Error in code	N	Error in generated code, however if there is an existing failing pipeline already the system automatically fixes it in the next run
8	Create GitLab Pipeline	V3	3	Y	-	42	-	-	Fixed the existing failing pipeline
9	Deploy to Kubernetes (Simulated)	V1	1	Y	-	306	-	-	-
9	Deploy to Kubernetes (Simulated)	V1	2	Y	-	270	-	-	-
9	Deploy to Kubernetes (Simulated)	V1	3	Y	-	202	-	-	-
9	Deploy to Kubernetes (Simulated)	V2	1	N	-	80	Error in execution	N	Similar to error in task 5 in generated the code and file but didn't add code (now attempt to fix bug)
9	Deploy to Kubernetes (Simulated)	V2	2	Y	-	91	Twice unexpected eof, once no such file	Y, 3 times	Now after adjusting the prompt to add code to files differently it worked
9	Deploy to Kubernetes (Simulated)	V2	3	Y	-	89	Again 4 similar errors but all recovered	Y, 4 times	Now after adjusting the prompt to add code to files differently it worked
9	Deploy to Kubernetes (Simulated)	V3	1	Y	-	43	-	-	Created the needed files
9	Deploy to Kubernetes (Simulated)	V3	2	Y	-	54	-	-	-
9	Deploy to Kubernetes (Simulated)	V3	3	Y	-	52	-	-	-

Figure B.3: Test Evaluation Table - Part 3: Tasks 7–9

10	Recover from Repo Already Cloned	V1	1	N	-	-	-	-	Task not supported by v1
10	Recover from Repo Already Cloned	V1	2	N	-	-	-	-	Task not supported by v1
10	Recover from Repo Already Cloned	V1	3	N	-	-	-	-	Task not supported by v1
10	Recover from Repo Already Cloned	V2	1	Y	-	11	Successfully detected the error	Y	Removed the cloned version and re-cloned it (can be good in case there were changes in the meanwhile)
10	Recover from Repo Already Cloned	V2	2	Y	-	12	Successfully detected the error	Y	Removed the cloned version and re-cloned it (can be good in case there were changes in the meanwhile)
10	Recover from Repo Already Cloned	V2	3	Y	-	10	Successfully detected the error	Y	Removed the cloned version and re-cloned it (can be good in case there were changes in the meanwhile)
10	Recover from Repo Already Cloned	V3	1	Y	-	6	-	-	Immediately found it out and marked task as complete
10	Recover from Repo Already Cloned	V3	2	Y	-	7	-	-	-
10	Recover from Repo Already Cloned	V3	3	Y	-	6	-	-	-
11	Simulated Merge Conflict	V1	1	N	-	-	-	-	Task not supported by v1
11	Simulated Merge Conflict	V1	2	N	-	-	-	-	Task not supported by v1
11	Simulated Merge Conflict	V1	3	N	-	-	-	-	Task not supported by v1
11	Simulated Merge Conflict	V2	1	Y	-	41	1 error (failed push)	Y	Successfully solved the merge conflict when pushing
11	Simulated Merge Conflict	V2	2	Y	-	55	3 errors (already cloned, failed push)	Y, 3 times	Successfully solved the merge conflict when pushing
11	Simulated Merge Conflict	V2	3	Y	-	41	1 error (failed push)	Y	Successfully solved the merge conflict when pushing
11	Simulated Merge Conflict	V3	1	Y	-	15	-	-	Successfully pulled latest changes before doing the task so no issue came up and conflict got solved automatically
11	Simulated Merge Conflict	V3	2	N	-	18	Bug where it doesn't execute the steps	N	Bug where system doesn't execute the steps after before successfully pushing latest changes from the repo
11	Simulated Merge Conflict	V3	3	Y	-	26	-	-	-
12	Pulumi Deployment to AWS EC2	V1	1	Y	-	13	-	-	-
12	Pulumi Deployment to AWS EC2	V1	2	Y	-	13	-	-	-
12	Pulumi Deployment to AWS EC2	V1	3	Y	-	12	-	-	-
12	Pulumi Deployment to AWS EC2	V2	1	Y	-	93	3 errors (unexpected eof)	Y, 3 times	Those errors are currently common when system is trying to add code
12	Pulumi Deployment to AWS EC2	V2	2	N	-	98	4 errors (unexpected eof)	Y, 4 times	Similar to error in task 5 in generated the code and file but didn't add code (again attempt to fix bug)
12	Pulumi Deployment to AWS EC2	V2	3	Y	-	92	3 errors (unexpected eof)	Y, 3 times	Those errors are currently common when system is trying to add code, even created a README (was not the case with DeepSeek or qwen)
12	Pulumi Deployment to AWS EC2	V3	1	Y	-	74	-	-	Created even a whole infrastructure folder with bunch of different files that are needed
12	Pulumi Deployment to AWS EC2	V3	2	Y	-	83	-	-	-
12	Pulumi Deployment to AWS EC2	V3	3	Y	-	68	-	-	-

Figure B.4: Test Evaluation Table - Part 4: Tasks 10–12