UNIVERSITY OF CALGARY

LLM-based MaSE -

A Software Development Framework for Developing Multi-Agent System

by

Sahar Hajjar Zadeh

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING

CALGARY, ALBERTA
DECEMBER, 2025

Abstract

The automation of software development prompted extensive research for decades. Recent advancements in AI, particularly large language models (LLMs), are driving transformative changes in the field. While LLMs have demonstrated impressive performance in code generation, challenges remain in generating complex and domain-specific code. To fully realize their potential, it is crucial to extend their application throughout the entire software development lifecycle. This necessity has driven us to explore and evaluate the role of these models in supporting all stages of the software engineering process. In this thesis, we propose an LLM-based framework tailored for the development of multiagent systems (MAS). By focusing on MAS and leveraging the MaSE methodology in conjunction with the JADE platform, our framework ensures the generation of accurate, traceable, and reliable software artefacts efficiently. The framework uses predefined prompts to guide the LLM through each step of the methodology, producing outputs that adhere to domain-specific requirements. Empirical evaluations on sample projects, through a case study and an experiment, demonstrate that this approach not only accelerates the development process but also results in artefacts that are more comprehensive and modular than those produced by human developers, showcasing the transformative potential of LLMs in software engineering.

Preface

This thesis was partially prepared with the assistance of generative AI tools in limited and appropriate ways. Specifically, ChatGPT was used for minor grammar editing and text refinement. All generative AI tools used in this thesis were released prior to June 2025. The intellectual contribution, research design, analysis, interpretation of results, and conclusions are entirely the author's own. All AI-assisted outputs were carefully reviewed and revised by the author to ensure accuracy, originality, and alignment with academic standards. The use of generative AI tools was authorized by the supervisor (Dr. Behrouz Far) and adheres to the University of Calgary Guidelines for Generative AI Use in Graduate Studies (April 2025) [1].

Acknowledgements

There are many people who supported me during the completion of this thesis, and I would like to take this opportunity to express my sincere gratitude to them.

I would like to express my deepest gratitude to my supervisor, Prof. Behrouz Far, for his invaluable guidance, continuous support, and encouragement throughout this research. His mentorship and approach to addressing challenges helped me grow not only as a researcher, but also as a person. I am also grateful to my co-supervisor, Dr. Zahra Shakeri, for her support and guidance during this work.

I would also like to express my appreciation to my friends and colleagues at the University of Calgary for their support and collaboration. In particular, I would like to acknowledge Yousef Mehrdad, Alireza Esmaeili, Somayeh Modaberi, Fatemeh Ghaffarpour, Esmaeil Shakeri, Anja Slama, Mehrnaz Senobari, Mehdi Eshragh, Mahmood Khalghollah, Mehregan Biglarbeiki, Zahra Safari, and Saba Farshbaf.

I am thankful to my previous supervisor at Sharif University of Technology, Dr. Raman Ramsin, for his mentorship during my earlier master's studies, which laid a strong foundation for my academic and research development.

I am deeply grateful to my family. I would like to thank my parents, my mother and my father, for their unconditional love and constant encouragement. I am especially thankful to my husband, Bahman, for his love and support throughout this journey, and to my daughter, Asal, whose love and joy were a constant source of motivation and strength.

Finally, I extend my sincere appreciation to the members of my examination committee, Dr. Mahmood Moussavi and Dr. Steve Drew, for their time, careful review, and valuable feedback, which helped improve the quality and clarity of this work.

To my husband, Bahman, for his love and support,

To my lovely daughter, Asal, my source of joy and inspiration,

To my mother and my father, for their endless love and strength,

To my brother, Ehsan, and my sister, Sanaz,

And to my lovely niece, Delva.

Table of Contents

List of Figures

List of Tables

List of Symbols, Abbreviations, and Nomenclature

Symbol	Definition
ACC	Agent Communication Channel
AMS	Agent Management System
API	Application Programming Interface
BDD	Behavior-Driven Development
CoT	Chain of Thought (prompting technique)
DF	Directory Facilitator
FIPA	Foundation for Intelligent Physical Agents
FSA	Finite State Automaton
GUI	Graphical User Interface
HumanEval	Hand-Written Evaluation Set
JADE	Java Agent DEvelopment Framework
LLM	Large Language Model
MaSE	Multi-agent Systems Engineering
MAS	Multi-Agent System
MBPP	Mostly Basic Python Problems
MTPB	Multi-Turn Programming Benchmark
PE	Prompt Engineering
Q-Q Plot	Quantile-Quantile Plot
RAG	Retrieval-Augmented Generation
RLHF	Reinforcement Learning from Human Feedback
RM ANOVA	Repeated Measure Analysis of Variance

SDLC	Software Development Life Cycle
SOP	Standard Operating Procedure
ULS	Universal Leveled Scale
UML	Unified Modeling Language
UI	User Interface

Chapter 1

Introduction

1.1 Introduction and Motivation

Software engineering (SE) involves designing, developing, testing, and maintaining software systems [5]. This process is highly complex and time-consuming, prompting extensive research into its automation over the years [6]. Recently, advancements in AI, particularly large language models (LLMs), suggest a significant transformation in software engineering. AI-augmented software development appears to be the future [7], promising automated processes and the creation of more accurate software systems in less time.

Large Language Models (LLMs) are advanced machine learning models trained on a large amount of data extracted from books, articles, code, repositories, and websites. They learned patterns and relationships in language, which made them capable of simulating human linguistic capabilities [7], [8]. Although like human intelligence, they face some challenges such as hallucination (generating false or fiction outputs) and nondeterminism (generating different responses to identical inputs), their impressive power in producing human-like text has led to major transformations in language processing and many other domains [8], [9].

Additionally, LLMs have demonstrated significant potential to transform the software engineering process. These models can accelerate development workflows, leading to higher-quality software systems [8]. While their ability to perform different software engineering tasks such as requirement engineering and design, code generation and completion, software testing, maintenance and document generation has been investigated, but most research to date has focused on specific, isolated software engineering tasks such as code generation, testing, and code repair [8], [9]. While LLMs have shown great performance in generating code, they face challenges in producing complex code [10].

To unlock their full potential, it is essential to extend their application across the entire software engineer-

ing lifecycle [8]. Hybrid techniques, traditional software engineering plus LLMs, has a key role in developing reliable and efficient LLM-based software engineering approaches [9]. This necessity has driven us to explore and evaluate the role of these models in supporting all stages of the software engineering process.

1.2 Problem Statement

As discussed in Section 1.1, while LLMs have the potential to transform the software engineering domain, their current applications in software engineering mostly focus on specific tasks in the implementation phase, such as code generation which is often limited to coding simple programming problems. The LLMs are excellent at generating a function or fixing a bug, but they fall short when asked to design an entire system. The crucial phases of analysis and design, which lay the foundation for the rest of the software, are often left unsupported.

To fully utilize the potential of LLMs, our research aims to bridge this gap by extending the use of LLMs across the entire software development lifecycle (SDLC). We created a structured framework that guides an LLM through every stage of development, from initial requirements all the way to final code.

1.3 Proposed Methodology and Thesis Contributions

This thesis presents a novel framework for LLM-based software engineering process tailored for multi-agent systems (MAS) development. By narrowing the scope to a specific type of software system (MAS), adopting a guiding methodology such as MaSE, and leveraging a platform like JADE, the framework aims to deliver more accurate and reliable software systems in less time.

The framework is based on two choices: focusing on the Multi-Agent Systems and using the MaSE methodology. Multi-Agent Systems are complex, distributed, and autonomous systems, making them a good candidate for testing full-lifecycle automation. More importantly, MaSE is chosen for the guiding methodology, which is a widely recognized approach for MAS development. MaSE methodology breaks down MAS development into distinct phases, each producing a specific, well-defined artefact.

Through the application of prompt engineering (PE) techniques, from few-shot prompting to more complex methods like Chain-of-Thought (CoT), we guide LLMs through the software development methodology's steps to generate software analysis artefacts, design models, and code.

Our work explores the potential of GPT-4, a state-of-the-art LLM, for generating software applications. As outlined, our primary focus is on developing MAS following the MaSE methodology, with JADE serving as the implementation platform. The proposed LLM-based framework leverages predefined prompts to

navigate the LLM through key steps of the methodology, ensuring the generation of domain-specific and methodologically sound outputs. Due to the lack of domain-specific datasets, we conducted empirical studies to evaluate the proposed framework.

Qualitative evaluation results through a case study indicate that the LLM-based MaSE methodology not only accelerates the development process but also produces more accurate artefacts while preserving traceability. While some implementation details are missing, the ability to generate initial artefacts and code in less than an hour provides a solid foundation for complex projects. This framework serves as a starting point, bringing us closer to the goal of automating software development.

Additionally, this research contributes to the challenging and unexplored area of LLM evaluation in software engineering. We conducted an experiment with statistical analysis to evaluate the performance of the LLMs following the proposed framework for multi-agent systems (MAS) development. A dedicated sub-experiment was performed for each step of the LLM-based MaSE methodology. The results indicate that, across all steps, LLM performance, in MAS development based on LLM-based MaSE methodology, was at least equivalent to human performance, while producing outputs within minutes.

In summary, this work demonstrates that LLMs have significant potential for automating the software development process by reducing the time and effort required to produce software artefacts and code. We believe that LLMs represent a key solution in realizing the vision of fully automated software development.

1.4 Organization of thesis

This thesis is structured as follows:

- **Chapter 2** reviews related work, providing the foundation for our study.
- **Chapter 3** introduces our proposed framework, detailing its components and methodology, and the supporting tool developed to implement it.
- **Chapter 4** presents the qualitative evaluation of the framework through a case study.
- **Chapter 5** describes an experiment conducted for performance evaluation and presents statistical analysis of the results.
- **Chapter 6** concludes the thesis and outlines potential future research directions.

Chapter 2

Background and Related Works

2.1 Introduction

In developing our framework, we made two choices: focusing on Multi-Agent Systems and adopting the MaSE methodology. This chapter provides the necessary background by introducing key concepts and reviewing related research efforts.

Section 2.2 presents an overview of Multi-Agent Systems (MAS). Section 2.3 introduces the MaSE methodology, followed by Section 2.4, which covers the JADE framework. Section 2.5 defines Large Language Model (LLMs), and Section 2.6 outlines common challenges in interacting with LLMs. Section 2.7 presents LLM customization techniques and Section 2.8 introduces prompt engineering (PE) as a technique for adapting LLMs to new tasks. Section 2.9 reviews research on LLM-based software engineering and code generation. Finally, Section 2.10 explores the integration of LLMs and MAS development.

2.2 Multi-Agent System (MAS)

Multi-Agent Systems (MAS) are distributed and autonomous systems composed of multiple interacting agents. These systems are particularly suitable for modeling complex, dynamic and domain-specific environments, making them ideal for testing full-lifecycle automation.

A multi-agent system consists of multiple agents that interact, often through message exchange, where each agent can perform independent actions on behalf of its user or owner to satisfy its design objectives [2]. As noted in [11], a multi-agent system typically comprises a group of loosely connected agents working within a shared environment to achieve a common goal.

Although there is no universally accepted definition for an agent [2], it is generally described as a flexible

and autonomous entity capable of perceiving its environment through sensors and acting upon it through actuators (see Figure 2.1). Unlike traditional systems, agents are characterized by their situatedness, autonomy, and goal-directed behavior [11].

Features such as efficiency, reliability, robustness, scalability, flexibility and reusability have contributed to the widespread use of MAS technology across application domains [11].

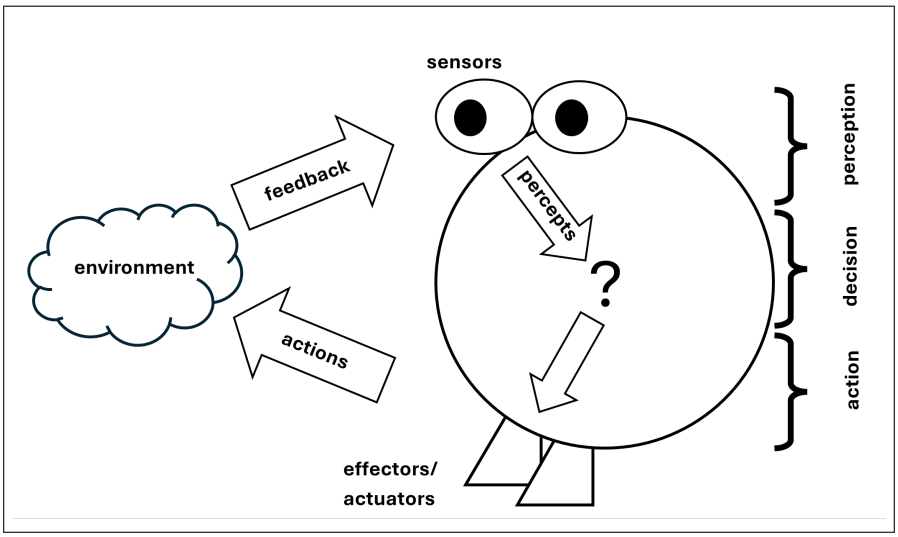

Figure 2.1: An agent in its environment [2]

As MAS technologies have gained attention in the computer science community, several development methodologies have emerged, including Gaia, MaSE and Tropos. These methodologies aim to support both the understanding and design of multi-agent systems. They typically consist of a set of models and accompanying guidelines to structure the development process [2].

2.3 MaSE Methodology

MaSE (Multi-agent Systems Engineering) is a widely used methodology for developing multi-agent systems (MAS). One of its main advantages is its structured approach, which includes a set of models and transformation steps that define how new models are derived from earlier ones. MaSE supports the full lifecycle of MAS development, covering analysis, design and implementation of heterogeneous MAS [3].

Built upon object-oriented techniques, MaSE defines a set of models derived from standard UML diagrams. It follows an iterative process and emphasizes traceability, allowing any object created during the analysis and design phase to be traced forward or backward within the methodology [3].

MaSE consists of two main phases: Analysis and Design. The analysis phase includes three steps, while the design phase includes four. In each step one or more models are created. An overview of the methodology

is shown in Figure 2.2.

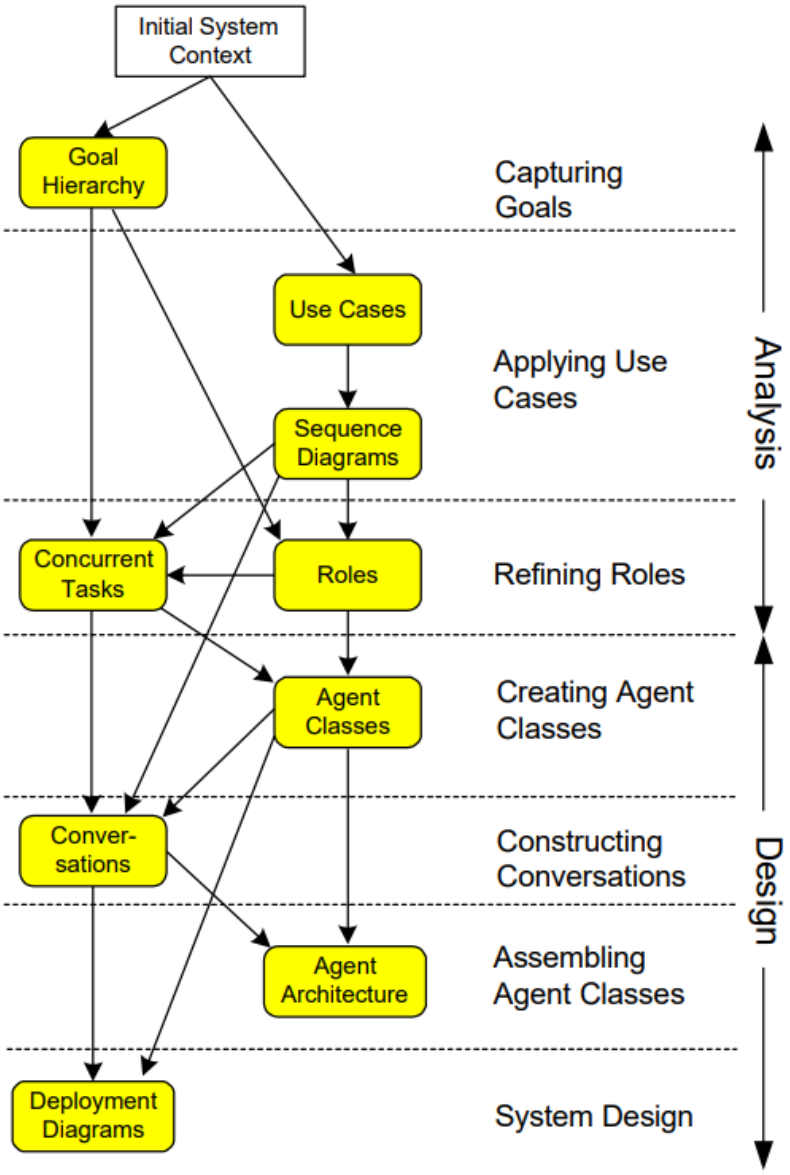

Figure 2.2: MaSE methodology [3]

Fundamentally, MaSE decomposes the complex task of MAS development into manageable phases and outputs. This structured decomposition aligns well with the operational needs of LLMs, as it transforms the abstract goal of "building a system" into a concrete series of tasks, making the entire process manageable and automatable.

For developing our framework, based on MaSE methodology's steps, we have derived a short direct instruction set and included key points to guide the large language model (LLM) in generating the artefacts

for each step.

In the following sections, we provide a brief overview of the specific steps involved in MaSE's analysis and design phases, based on [12].

2.3.1 Analysis phase

The main objective of the analysis phase is to define the system's goals, roles and associated tasks.

Capturing Goals

System goals are derived from requirements and organized in a Goal-Hierarchy, a directed acyclic graph, where nodes represent goals and edges define goal-subgoal relationships. The process starts by identifying the overall system goal and decomposing it into subgoals. Four types of goals are defined: summary, non-functional, combined and partitioned.

1. **Summary Goals:** These represent an overall goal, often reflecting the primary system goal, and are typically composed of several related peer goals.
2. **Non-functional Goals:** These goals don't directly contribute to functional aspects of the system but are critical for the system operation.
3. **Combined Goals:** These group together multiple very similar or identical goals. This usually happens when a goal is subgoal of more than one parent goal.
4. **Partitioned Goals:** These are fully achieved when all their subgoals are satisfied. They can be excluded from further analysis.

Applying Use Cases

Use cases describe system behavior to help identify roles and responsibilities. They are modeled using sequence diagrams (similar to UML), where interactions are defined between roles.

Refining Roles

Based on the goals and use cases, a role model is created that specifies roles, their assigned tasks, and relationships. Tasks are then detailed in concurrent task models, which are infinite-state automata that describe both internal agent behavior and transitions between tasks or agents.

2.3.2 Design phase

Creating Agent Classes

Agent classes are derived from the roles defined in the analysis phase. An agent class diagram is produced, showing agents, the roles they perform, and conversations between them.

Constructing Conversations

Conversations define coordination protocols between agents. Each is modeled using two communication class diagrams, one for the initiator agent and one for the responder agent. Conversation begins when the initiator sends a message. Upon receiving it, the responder checks whether it matches an active conversation. If a match is found, the conversation transitions to a new state and the responder performs any required actions; otherwise, the message is treated as a new conversation request. The responder compares the new message to the message type it can respond to and starts a new conversation if there is a match.

Assembling Agents

This step involves defining the agent architecture and its components. Designers can use predefined architecture and components or create new ones. Each component includes attributes and operations and may include sub-architectures. At a minimum, there should be an operation for any action or activity in the communication class diagram.

System Design

The final step is to create a deployment diagram for the system that specifies the number, type and location of agent instances. This diagram provides important information, such as hostnames and addresses for communications, needed for implementation. Designers must also account for communication and processing requirements, adjusting for factors such as computational power and network bandwidth.

2.4 JADE Framework

JADE (Java Agent Development Environment) is a software framework designed to support the development of multi-agent systems (MAS). It is an open-source project that aims to simplify the creation of MAS by adhering to the FIPA (Foundation for Intelligent Physical Agents) specifications for intelligent MAS [13].

JADE provides several built-in features that facilitate agent development. It provides a FIPA-compliant platform where essential agents like the Agent Management System (AMS), the default Directory Facilitator

(DF), and the Agent Communication Channel (ACC) are automatically initialized at startup, supporting distributed agent environments and multi-domain configurations. The AMS acts as the platform manager, ensuring that each agent has a unique name. The DF provides a Yellow Pages service, allowing the agents to find each other. The ACC manages communication between agents, both within the platform and with external agents [13].

JADE also offers a Java API for messaging, efficient intra-platform communication using Java object serialization, and support for agent mobility. Additionally, it includes libraries for interaction protocols, ontology management, and a graphical interface for monitoring and managing agents and platforms [13].

2.5 Large Language Models (LLM)

Recent advances in machine learning and deep learning have led to the development of large language models (LLMs), which are neural networks trained on vast amounts of data [7]. These models can generate human-like text that is often grammatically correct but may sometimes lack semantic accuracy. LLMs have been employed for a wide range of tasks, such as translation, summarization, question answering and software engineering tasks like code completion and bug reporting [7].

LLMs, also known as pre-trained language models, can be categorized into three groups based on their internal architecture: encoder-only, encoder-decoder, and decoder-only models. Each category is optimized for specific types of tasks in natural language processing (NLP) based on how information is processed and generated. In LLM architecture, the encoder module maps the input of a specific type to a hidden vector space, while the decoder module maps the hidden vector space back to the original type [8], [9].

- **Encoder-Only LLMs**

  Encoder-only models are composed solely of the encoder module, designed to process and encode input effectively. They excel in understanding input structures and are well-suited for tasks like code completion and bug reporting in the software engineering (SE) domain. An example of this architecture is BERT (Bidirectional Encoder Representations from Transformers) [8], [9].

- **Encoder-Decoder LLMs**

  Encoder-decoder models incorporate both encoder and decoder modules. They are proficient at understanding input and output structures, capturing semantics, and generating output with accurate structure. In the SE domain, these models are ideal for complex, multifaceted tasks such as code translation and code summarization. Examples include T5 (Text-to-Text Transformer) and BART (Bidirectional and Auto-Regressive Transformers) [8], [9].

- **Decoder-Only LLMs**

  Decoder-only models, such as GPT (Generative Pre-trained Transformers), are designed exclusively with a decoder module. These models specialize in generating output sequences based on input context, making them particularly effective for tasks like code generation and code completion [8], [9].

Studies indicate that the decoder-only LLMs perform best for generating tasks, which aligns with the primary objective of our framework [8], [9].

2.6 LLM Challenges

Despite their impressive capabilities, LLMs performance in generation tasks are subject to several challenges:

1. **Nondeterminism:**

   One of the primary challenges with LLMs is their nondeterministic nature, meaning that the same prompt can produce varying responses across different runs. Furthermore, minor changes in the prompt may lead to significantly different outputs, assuming that the temperature parameter, which controls the randomness of LLM responses, remain at the same non-zero value [9]. To assess the robustness of LLM-based approaches, experiments often involve repeating the same process multiple times and selecting the best output to mitigate the effects of non-determinism and improve the overall quality of results [14].

2. **Hallucination:**

   Much like human intelligence, LLMs are capable of fictional or inaccurate outputs, a phenomenon known as hallucination. While these outputs may seem plausible, they can be misleading or entirely untrue. In some cases, in the context of software engineering, hallucination can sometimes be beneficial by suggesting useful improvements. However, hybrid approaches, combining traditional software engineering methods with LLM support, are generally more effective in mitigating the risks associated with hallucinations [9].

2.7 LLM Customization Techniques

Several techniques can be employed to customize the LLM response or guide pre-trained models to deliver more precise and contextually relevant results:

1. **Fine-Tuning:** This involves retraining a pre-trained model using domain-specific or task-specific data to enhance its performance for specialized applications [15].

2. **Prompt Engineering (PE):** This method focuses on crafting optimized queries to elicit more accurate and relevant responses from LLMs by refining the input. Well-structured prompts can significantly enhance the model's ability to understand context and deliver refined outputs [15].

3. **Retrieval-Augmented Generation (RAG):** To improve output quality, RAG integrates external information by retrieving relevant documents from external databases and using them as context for content generation. This approach enhances the quality of LLM output and is particularly effective in scenarios where the LLM lacks specific information or data [16].

4. **Reinforcement Learning (RL):** Human trainers provide feedback, either positive or negative, on the model's generated responses. This feedback is then used to further refine and optimize the LLM [17]. This iterative learning process, often referred to as Reinforcement Learning from Human Feedback (RLHF), allows LLMs to adapt and improve based on evaluative guidance [18].

RAG and RL are often used alongside prompt engineering techniques to enhance performance and relevance of LLM responses. Among the various customization techniques, fine-tuning and prompt engineering are the most discussed approaches. When comparing these two methods, prompt engineering stands out for its cost-effectiveness, ease of use and faster result delivery. Although neither approach always outperforms the other in terms of accuracy, prompt engineering appears to be the winner in the long term [15].

In summary, the choice of LLM architecture and customization techniques depends largely on the intended application and the specific requirements of the task. Decoder-only models currently lead in generative tasks, while prompt engineering remains a cost-effective strategy for optimizing model outputs.

2.8 Prompt Engineering

The prompt plays a critical role in shaping the responses generated by large language models (LLMs) [19]. The quality of the response is directly influenced by the quality of the prompt, with human-crafted prompts often resulting in the best output [20]. Using prompt engineering techniques, LLMs rely entirely on their embedded knowledge and the input provided through the prompt to perform the requested task [21]. Considering certain key principles, we can create effective prompts that enable LLMs to produce high-quality responses [17].

Various studies have contributed to the development of well-defined prompts, a process known as prompt engineering. This technique, which has gained significant attention recently, tailors LLMs for specific tasks.

Among the approaches available, notable techniques include [20], [21], [14]:

- **Zero-shot prompting:** Involves querying the LLM with carefully constructed prompts without providing any examples of the desired task. The model relies entirely on its pre-trained knowledge to respond.
- **One-shot/Few-shot:** Involves presenting the LLM with one (one-shot) or a small number (few-shot) of examples within the prompt. The model can learn from the provided examples to perform the new task.
- **Chain-of-Thought (CoT):** This technique guides the LLM through a step-by-step reasoning process by including a series of logically interconnected prompts. This helps the model to generate more coherent and accurate responses for complex tasks.
- **Self-consistency:** A technique that uses prompts to guide the language model toward producing responses that align with its earlier outputs. The goal is to maintain logical coherence and avoid contradictions across different parts of the model's answer.
- **Reasoning and Acting (ReAct):** A technique that uses prompts to guide the model in reasoning through a given situation and then generating appropriate responses based on that reasoning process.

In addition to these techniques, some studies have introduced collections of prompt patterns, which are successful approaches for constructing prompts. These patterns help users to customize the output and interactions of LLMs effectively [22]. For example, [22] and [23] presented catalogs of prompt pattern specially designed for software engineering tasks. Such patterns enable users to better define inputs, context, and outputs, improving the overall prompt quality.

Furthermore, [20] introduced a five-step process for constructing effective prompts:

1. **Defining the Goal:** Establishing a clear objective is essential first step in creating a well-structured prompt.
2. **Understanding the Model's Capabilities:** Recognizing the model's strengths and limitations allows for better alignment with its capabilities.
3. **Choosing the Right Format:** Using a clear and concise format helps the model to understand the request accurately.
4. **Providing Context:** Including relevant context enables the model to generate more accurate and relevant responses.
5. **Testing and Refining:** Iteratively testing and refining the prompt is important to enhance its effectiveness before deploying it in production.

2.9 LLM-based Software Engineering

LLM-based software engineering (LLM-based SE) refers to programs where the LLMs are utilized in artefacts production or software development process [9]. Since the emergence of LLMs, researchers in software engineering, like other fields, have attempted to leverage this new technology. Ever since, LLMs have been applied and evaluated across various software engineering tasks, including requirement analysis, program analysis, code generation, program repair and software testing. Additionally, some studies have explored the practical application of LLM in these tasks through empirical experiments [8].

Research in LLM-based SE can be categorized according to the Software Development Life Cycle (SDLC). The SDLC is a widely adopted framework in software engineering that outlines the process of planning, creating, testing, and deploying software systems. Different authoritative sources outline different phases in the SDLC, typically include:

1. Analysis (Requirement Engineering)
2. Design
3. Implementation (Coding)
4. Testing
5. Deployment
6. Maintenance

Current studies show that most efforts in LLM-based SE have concentrated on code generation, code testing, and code repair tasks aligning with the implementation, testing, and maintenance phases of SDLC. However, phases such as analysis (requirement engineering) and design have received comparatively less attention [8], [9].

In LLM-based SE, code generation has been the primary focus. Several studies have undertaken fine-tuning baseline LLMs to develop specialized models that are specifically trained for code generation. Notable examples include CodeX [24], InCoder [25], CodeGen [26] and CodeLlama [27]. CodeX [24] is a fine-tunned GPT model trained on GitHub code. Its evaluation is largely based on the HumanEval dataset, which primarily consists of isolated coding problems that require implementing simple algorithms rather than tackling real-world software requirements. So, while HumanEval tests basic algorithmic and language comprehension skills, it is far from real software engineering. A true software requirement would involve problem analysis, requirement elicitation, architectural design, and iterative development, all of which are missing here. InCoder [25] is a generative model designed for code infilling and synthesis, trained on licensed

code from GitHub and GitLab. It focuses on completing partially written code. CodeGen [26] introduced a different approach with its focus on multi-turn program synthesis. Unlike traditional code generation models that generate an entire program in a single pass, CodeGen can break down the problem into multiple steps, making the synthesis process more interactive and structured. This model is evaluated using the Multi-Turn Programming Benchmark (MTPB), which requires models to synthesize a program step-by-step. While the evaluation dataset seems different than other datasets like HumanEval, it also contains standalone programming problems, not full software engineering tasks that are broken down to several steps. Similarly, CodeLlama [27] is a family of models trained and evaluated on major benchmarks like HumanEval and MBPP, further demonstrating LLM capabilities in standard programming tasks.

Most LLMs are trained on public datasets such as HumanEval and MBPP, which consist of natural language and code pairs representing programming problems, corresponding solutions, and, in some cases, test cases. However, public datasets sourced from the internet may not always contain accurate data. To address this, Phi-1 [28] was introduced as a transformer-based model trained for code generation. Unlike traditional large-scale models, phi-1 focuses on high-quality training data from textbooks. The study demonstrates that high-quality data selection can dramatically improve performance, allowing phi-1 to outperform much larger models. The model evaluated on datasets such as HumanEval and MBPP. Similarly, [29] presented AlphaCode, a model trained on competition-level problems from GitHub, evaluated using CodeContests dataset. Although more complex than simple input-output coding pairs, it still consists of isolated programming problems rather than real software engineering tasks.

Beyond general-purpose code generation, some studies have focused on domain-specific applications. GenLine [30] is a natural language code synthesis tool designed for domain specific code generation (e.g. web, game). It allows users to input natural language descriptions to generate or modify code through task-specific prompts, offering an experience similar to GitHub Copilot with some differences in style and approach. DomCoder [31] extends this concept by specializing in function block code completion for specific domains, evaluated using domain-specific datasets extracted from GitHub.

There are also conceptual studies exploring LLMs for higher-level tasks, such as generating Angular components from Behavior-Driven Development (BDD) specifications [32]. While the idea is innovative, the study lacked an actual implementation, dataset, and evaluation, stopping at the conceptual stage without proof-of-concept.

Despite their strengths in code generation, LLMs face challenges in handling complex code generation tasks [6]. To tackle this, some studies have been inspired by human behavior in solving complex tasks during the software development process. For example, [10] proposed an iterative framework comprising four phases that are performed by LLMs acting as analyst, designer, developer, and tester. This approach improved code

quality compared to direct generation methods, though it still primarily focused on programming problems using public datasets like HumanEval and MPBB.

Another study [33] proposed a methodology inspired by human planning, where LLMs first generate a high-level plan based on the user intent before producing the actual code. This structured approach simulates human problem-solving by planning before execution, leading to more coherent code generation.

Although LLMs have shown significant promise in generating code from natural language descriptions, they remain limited to solving isolated programming problems rather than delivering full software engineering artefacts which are crucial to the development process and should not be overlooked. Most research to date has focused on code generation and completion, overlooking critical SDLC phases like Requirement Engineering and Design. To fully leverage LLMs for software engineering, future research must expand beyond isolated code synthesis to include a broader range of development tasks, including analysis and design.

2.10 LLM-based MAS Development

This section explores the studies that utilized LLM with MAS frameworks. These studies differ totally from our work in terms of both goals and methodologies. In existing literature, researchers leveraged LLM-powered agents to address software engineering tasks through intelligent agent-based architecture.

For example, [34] and [35] introduced multi-agent systems for automated software improvement and automated code review, respectively. Their focus was on optimizing individual software processes rather than integrating LLMs throughout the entire development lifecycle.

In [36], the authors introduced CodePori, an LLM-augmented multi-agent system designed for code generation. Although the framework demonstrated the capability to handle complex code generation, its underlying process remained somewhat unclear, and the results were evaluated using public datasets with relatively simple programming problems.

Similarly, [37] presented a unified platform using LLM-based agents to automate the conversion of requirements into structured deliverables. While their system attempted to generate multiple software artefacts beyond just code for web applications, they faced challenges related to accuracy and consistency between deliverables.

In [38], the Standard Operating Procedures (SOPs) was integrated into LLM-based agent collaboration for structured software development. They assigned specific roles (e.g. Product Manager, Architecture, Engineer, QA) to agents, mimicking real-world SDLC workflows. The SOP-guided communication between agents helps reduce hallucinations and errors, improving software quality and consistency.

The study in [39] introduced a software development framework where LLM-powered agents collaborate to complete different tasks. These agents follow a structured chat chain for communication, breaking down tasks into subtasks across design, coding, and testing phases. To reduce hallucinations, agents request additional clarifications before responding, ensuring more accurate and reliable software generation.

Finally, [40] proposed a software development framework that integrates prompt engineering with human feedback throughout the entire process. The development starts with generating high-level requirements, which are refined through user interaction. The system then proceeds with structure design, automated coding, and iterative testing. Finally, the human validates the system, updates requirements if needed, and the process repeats until a satisfactory implementation is achieved.

In Summary, while existing studies focus on LLM-powered agents to handle isolated software tasks, our approach extends LLM integration throughout the entire development process, aiming for more complete and consistent software artefacts.

Chapter 3

Proposed Framework and Tool

3.1 Introduction

In this chapter, we presented our AI-augmented MaSE framework, referred to as LLM-based MaSE, for developing multi-agent systems (MAS). The main goal of our framework is to utilize large language models (LLMs) to automate the MAS development process, starting from system requirements described in natural language. As discussed in Chapter 2, MaSE supports the entire lifecycle of MAS and includes a set of models that can be derived from earlier ones following the methodology transition steps. These features make MaSE a suitable foundation for the framework presented in this work.

Building on the MaSE methodology, we employ prompt engineering (PE) to ensure the LLM is context-aware and effectively guided through each stage of the development process. The most important part of our methodology is the creation of concise and clear prompts. Based on MaSE, we have developed a set of straightforward instructions based on the methodology steps, highlighting essential points to guide the LLM in generating the necessary models. Therefore, our approach extends LLM integration throughout the entire development process, aiming for more complete and consistent software artefacts.

In this chapter, we first outline the structure of our framework's prompts, then delve into the detailed steps of the development process. Lastly, we describe the application developed to support the framework.

The developed application plays a vital role in implementing the proposed framework, facilitating the testing and validation process. Although the application is basic, it is essential for testing and verifying the methodology, without which the validation could not have been completed effectively.

3.2 Prompt Structure

As described in Chapter 2, prompt engineering has different techniques, patterns and guidelines that help in constructing effective prompts. Following these guidelines, and understanding that prompts act as instructions containing input, context and output indicators [20], we developed a structured approach to prompt construction. This structure ensures clarity, precision, and relevance in the responses generated by the LLM. This structure includes five key elements:

1. **Instructions:** This section specifies the task the LLM should perform in the current step, along with some details on how to approach it.

2. **Important points:** This section highlights concise, direct points LLM should consider in generating the output. These points extracted from an iterative refinement process and may include details like the number of models to generate, traceability requirements, or adherence to specific output templates with placeholder replacement.

3. **Traceability Links:** This section defines how the step relates to models or data generated in previous steps.

4. **Output Template:** This section presents the template that the LLM must follow when generating output, incorporating the "Template" pattern from [22]. This ensures consistency and prevents the LLM from adding unnecessary information.

5. **Example:** This section provides a sample output based on a Conference Management System to illustrate the expected format and content. Including such examples helps the LLM learn both the syntax and semantics of the desired output, serving as a reference for generating responses that align with the required structure and meaning.

3.3 Constructing Prompts

Our framework provides a series of interconnected prompts that guide the LLM through each step of the MAS development process, from requirements to analysis, design, and implementation. This step-by-step structure enables the model to build on previous outputs, effectively applies a form of Chain-of-Thought (CoT) prompting, where reasoning is broken down into manageable and traceable steps. Additionally, providing an example for each step serves as a form of one-shot prompting, helping the LLM to understand and replicate the desired output format and semantics.

By adhering to the structured approach for prompt construction, we developed the core of our framework, creating a prompt for each step in the MaSE methodology. These prompts specify what the LLM should do at each step, the important points to consider during the generation process, and the inputs required for the current step. Additionally, we determined output templates to guide the LLM in generating results, which helped in preventing the LLM from generating extra, non valuable details. This approach ensures the prompts guide the LLM effectively, resulting in more accurate and context-aware responses.

Constructing prompts was an iterative process. The final set of prompts were refined through a test-and-refine loop. In each iteration, we focused on making the "instructions" as clear and direct as possible. We also added new points to the "important points" section to prevent the LLM from repeating mistakes. For example, we included reminders to the LLM to avoid leaving placeholders in the output template and to replace them with the correct data.

Since LLMs generate data in text format (and their responses in image format are not yet as good as text), we aimed to improve the readability of the generated models by asking the LLM to output results in PlantUML format. PlantUML is an open-source tool that generates UML diagrams from simple text descriptions. We considered it a good option for generating models in text format that could be easily converted into visual diagrams. While this approach resulted in more readable models for documentation purposes, the LLM did not always generate syntactically correct PlantUML code. We identified two reasons for this: firstly, PlantUML does not inherently support MaSE models and require customization; secondly, the LLM's tendency to generate inaccurate or irrelevant content (hallucination) led to the issue.

Given these challenges, we decided to remove the complexity of generating PlantUML code. Instead, we focused on providing simple output templates that directly aligned with the instructions and commands, ensuring a clearer and more efficient model development process.

3.4 LLM-based MaSE

After defining the prompt structure, we proceed to develop our methodology. Building on the phases and artefacts of the MaSE methodology, our LLM-based approach constructs tailored prompts at each step to guide the LLM in navigating the methodology and generating the required models.

At the start of the methodology, we employ the "Persona" pattern from [22] to ask the LLM to play the role of a "Software Engineer expert in MAS development". Additionally, we provide the LLM with a brief explanation of the prompt structure used throughout the process. The initial prompt, also known as setup prompt, is as follows:

Act as Persona Software Engineer expert in multi-agent systems (MAS) development. We are going to work as a team to develop a MAS following MaSE methodology steps and using JADE platform. The first five steps are part of the analysis phase, the next four steps belong to the design phase, and the remaining steps are part of the system implementation phase. Each step includes:

- *Instructions – Task objectives and process*
- *Important Points – Key criteria to apply*
- *Traceability Links – How the step relates to prior outputs*
- *Output Template – Format and sample structure*

There are some important things you must remember during the development process:

- *When exact matches to previous artefacts are requested, ensure they are precise.*
- *Don't delegate any model or code generation to me, and don't ask whether to generate more — always return the full requested output.*
- *The focus is on functional system requirements. Please ignore non-functional requirements.*

Following this initial step, the process begins by gathering key project information from the user, including the project name and system requirements. This information forms the foundation for the first prompt, which asks the LLM to refine and list the system requirements. This step ensures that the LLM has a clear understanding of the desired requirements and establishes a basis for improved traceability across subsequent steps.

LLM-based process (Figure 3.1) is as follows:

1. The initial setup prompt asks the LLM to take the role of persona.
2. The user is prompted to provide the target system name and requirements to construct the first conversational prompt.
3. The iterative process begins with:
   (a) Setting the current step
   (b) Assembling the current prompt

(c) Prompting the LLM and obtaining the results

(d) If this is not the last step of the process, return to step 3

4. Finished

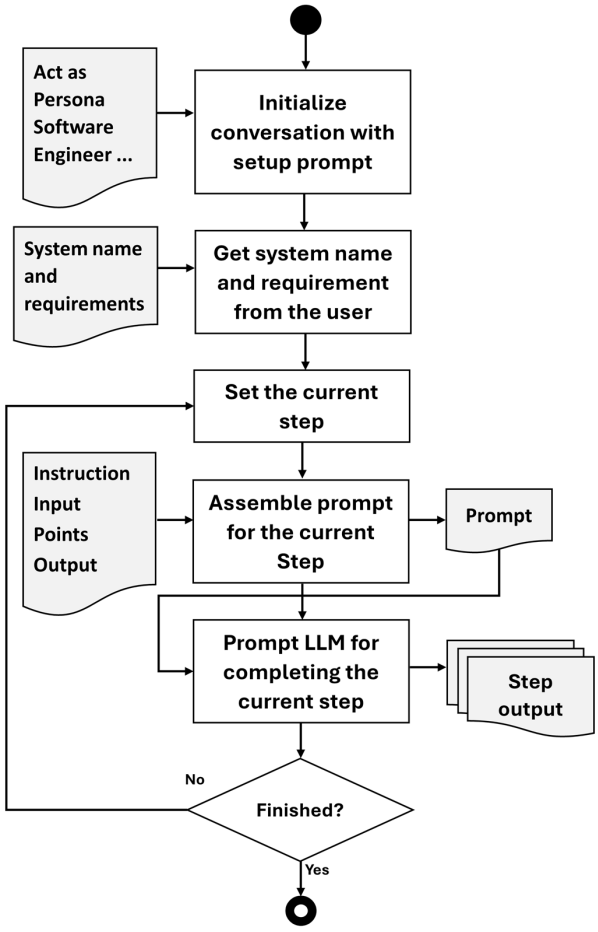

Figure 3.1: LLM-based MaSE process

The methodology then progresses through a series of steps, each prompting the LLM to generate specific artefacts. The first five steps correspond to analysis phase of the methodology, and the subsequent four steps are part of the design phase. Below is the prompt list for each step of the LLM-based MaSE.

3.4.1 Step 1. Requirement List

In the first step, the LLM is prompted to review and refine the system requirements provided by the user, generating a list of requirements in numeric format.

Instructions:

Please review, edit, and complete the functional system requirement list for developing the <system name>MAS application with the following initial requirements: <system requirements>

Important points:

1. *Clarify ambiguous points.*
2. *Add essential functional features or constrains only that I may have been overlooked.*
3. *Ensure consistency in language and formatting.*
4. *Do not include non-functional requirements such as performance, security, or usability constraints.*

Traceability Links:

All refined requirements must directly map to system functionalities and goals. Each requirement should represent a discrete function or service the system must provide. Each requirement should be clear enough to be decomposed into goals and roles in the subsequent MaSE steps. Please confirm that your generated requirement list covers all functional system requirements.

Output Template:

The refined system requirement list should follow the structured template below:

R1. `[Requirement1]`

R2. `[Requirement2]`

R3. `[Requirement3]`

R4. And so on

3.4.2 Step 2. Goal Hierarchy

This step marks the beginning of the MaSE analysis phase. Here, the LLM is prompted to generate the system's goal hierarchy based on the requirement list from previous step. Important points are added to ensure that the LLM creates a direct and acyclic graph with one global system goal and tags partitioned

goals.

Instructions:

Determine the system's goals and structure them in a directed acyclic graph (DAG) known as Goal Hierarchy Diagram. This process involves two main steps:

1. *Identifying Goals: Extract high-level goals from the system requirements and define their relationships.*
2. *Structuring Goals: Organize these goals hierarchically, decomposing them into sub-goals that represent different layers of abstraction*

Important points:

1. *Focus solely on what the system should accomplish, not on how it should be implemented. The hierarchy should reflect the system's core functionality and not implementation-specific details.*
2. *Identify partitioned goals—these are goals that are fully satisfied when all their sub-goals are achieved, without requiring additional tasks. These should be marked explicitly.*
3. *Avoid using placeholders; instead, provide clear and descriptive goal names based on system requirements.*

Traceability Links:

Each goal should be linked back to specific system requirements to verify completeness and consistency. Please confirm that your goals cover all functional requirements in the requirement list.

Output Template:

A structured Goal Hierarchy Diagram, formatted as follows:

G1. Main Goal (partitioned, if applicable)

G1.1. Sub-Goal

G1.1.1. Sub-Sub-Goal

G1.2. Sub-Goal

G1.3. Sub-Goal

G1.3.1. Sub-Sub-Goal (partitioned)

Example:

For a Conference Management System with the following requirement list:

R1. Authors should be able to submit their papers electronically to a conference paper database system. During the submission phase, authors should be notified of paper receipt and given a paper submission number.

R2. After the deadline for submissions has passed, the papers will be divided among the program committee (PC), who has to review the papers themselves or by contacting referees and asking them to review a number of the papers.

R3. Reviewers should be able to get papers directly from the central database and submit their reviews to a central collection point.

R4. After the reviews are complete, a decision on accepting or rejecting each paper must be made.

R5. After the decisions are made, authors are notified of the decisions and are asked to produce a final version of their paper if it was accepted.

Goals might be organized as follows:

G1. Produce Conference Papers (partitioned)

- *G1.1. Make Papers Available (partitioned)*
  - *G1.1.1. Collect Papers*
  - *G1.1.2. Distribute Papers*
    - *G1.1.2.1. Distribute Abstracts*
- *G1.2. Assign Reviewers to PC Members (partitioned)*
  - *G1.2.1. Partition Papers*
  - *G1.2.2. Assign Reviewers*
- *G1.3. Review Papers (partitioned)*
  - *G1.3.1. Prepare Reviews*
  - *G1.3.2. Submit Reviews*
- *G1.4. Collect Reviews*
- *G1.5. Select Papers*
  - *G1.5.1. Inform Authors*

3.4.3 Step 3. Sequence Diagram

In this step, the system's use cases are generated as sequence diagrams. The LLM is prompted to create positive and negative system use cases, displaying roles and communications between them.

Instructions:

Generate a full set of sequence diagrams, each being either a positive or negative use case, so that all goals are covered. These diagrams should represent specific scenarios that illustrate how roles interact to achieve system goals.

Important points:

1. *One Sequence Diagram per Use Case:*
    - *Each sequence diagram should correspond to a unique use case scenario representing a single, meaningful goal from the actor's perspective.*
    - *use cases should have a clear system outcome. Avoid overly broad diagrams (e.g., "Manage System") or overly detailed ones (e.g., "ConfirmDelivery").*
    - *Positive use cases depict normal interactions, while negative use cases illustrate failures or exceptions (e.g., task reassignment, missed deadlines, etc.).*
2. *Clearly Defined Roles:*
    - *Identify all relevant roles for each sequence diagram.*
    - *These roles must correspond to those identified in the Goal Hierarchy.*
    - *Do not include a generic "System" role.*
3. *Event-Based Communications:*
    - *Represent interactions explicitly as events/messages passed between roles.*
    - *Focus on the goal of the use case, not the implementation details.*
4. *Preconditions and Initial Actions:*
    - *Assume necessary preconditions are met.*
    - *Start with a clear trigger that initiates the process of the use case.*
5. *Handling Alternative Flows:*

- *For negative scenarios, include alternative flows or error-handling mechanisms (e.g. handling communication timeouts, retry mechanisms).*

6. *Time-Based Events:*

   - *When applicable, illustrate time-sensitive events like deadlines, reminders, or scheduling triggers.*

7. *Complete Interaction Flow:*

   - *Ensure each diagram shows the start, message exchanges, and completion of the interaction.*

8. *Avoid Placeholder Variables:*

   - *Use descriptive role and event names directly derived from system requirements.*

Traceability Link:

All sequence diagrams must trace back to and completely cover all goals in the Goal Hierarchy, ensuring every scenario directly contributes to achieving one or more goals. Confirm that all goals have satisfied by generated sequence diagram set.

Output Template:

The output should follow the structured template below:

Sequence Diagram: [Use Case Name]

Roles:

R1. Role A

R2. Role B

R3. Role C

Communications:

C1. Role A → Role B: Message M

C2. Role B → Role C: Message N

C3. Role C → Role A: Message O

Example:

For the Conference Management System, a sample use case for Paper Submission might be represented as:

Sequence Diagram: Paper Submission

Roles:

R1. Author

R2. PaperDB

Communications:

C1. Author → PaperDB: SubmitPaper()

C2. PaperDB → Author: ConfirmReceipt()

3.4.4 Step 4. Role Model

For this step, the LLM is prompted to generate a role model based on the system's goals. The model ensures that each role is linked to the appropriate goals, preserving traceability and context.

Instructions:

Create a Role Model to represent the roles, their assigned tasks, and the communication between these tasks, which are communication protocols, with an arrow pointing from the initiator to the respondent.

Make sure task granularity is appropriate: each task should be goal-driven, coherent, and executable by a role (sometimes by interacting with other roles).

Important points:

1. *Role Definition and Goal Satisfaction:*
    - *Every non-partitioned goal in the system's Goal Hierarchy must be satisfied by at least one role.*
    - *Each role should be designed to handle specific tasks that contribute to achieving its associated goals.*
    - *Ensure tasks are well-aligned with the role's goals and are neither too broad (e.g., "Manage System") nor too fine-grained (e.g., "Parse Input Format").*
2. *No Task Duplication:*
    - *Tasks should not be repeated across different roles. If multiple roles need to access the same logic, it should be abstracted into a shared role or utility.*

3. Interface Roles for External Access:

- *If a role needs to interact with external resources (e.g., databases, notification systems), a dedicated interface role should be created (e.g. a Database role for Database operations or Notifier for sending alerts).*

4. UI Separation for Human Interactions:

- *For any role with human interactions, define an explicit UI role separate from business logic (e.g. If you have a Buyer and Seller roles for business processes, also create a Buyer_UI and Seller_UI roles for UI interactions with human). This keeps business logic decoupled from the interface layer.*

5. Direct Reference to System Goals:

- *Each role in the model should explicitly reference the exact goals it is responsible for, using the names defined in the Goal Hierarchy.*

6. Task-Based Communication (Not Role-Based):

- *Describe communications protocols between tasks, not between roles.*
- *Focus on message-driven logic between tasks that represent role-level behavior.*

7. Avoid Placeholders:

- *Use concrete names for roles, tasks, and messages. Avoid generic placeholders.*

Traceability Link:

The role model should trace back to system goals and show clear mappings of which role is satisfying which goals. This should be evident through role-task assignments and inter-task communications. Please confirm that your generated roles and tasks and their communication covers all system goals and use cases.

Output Template:

The role model should follow the structured template below:

Roles and their assigned tasks:

R1. Role Name (List of Goals Satisfied - exactly match to the goals in goal hierarchy)

T1.1. Task 1

T1.2. Task 2

R2. Another Role Name (List of Goals Satisfied - exactly match to the goals in goal hierarchy)

T2.1. Task 3

T2.2. Task 4

Communication Between Tasks:

C1. Task 1 → Task 2: Message 1

C2. Task 2 → Task 3: Message 2

C3. Task 3 → Task 4: Message 3

Example:

For a Conference Management System, the role model could be represented as follows:

Roles and their assigned tasks:

R1. PaperDB (G1.1.1. Collect Papers, G1.1.2. Distribute Papers, G1.1.2.1. Distribute Abstracts)

T1.1. CollectPapers

T1.2. DistributePapers

T1.3. GetAbstracts

R2. Partitioner (G1.2.1. Partition Papers)

T2.1. PartitionPapers

R3. Assigner (G1.2.2. Assign Reviewers)

T3.1. AssignToReviewers

R4. Reviewer (G1.3.1. Prepare Reviews, G1.3.2. Submit Reviews)

T4.1. NegotiatePapers

T4.2. ReviewPapers

R5. Collector (G1.4. Collect Reviews)

T5.1. CollectReviews

R6. DecisionMaker (G1.5. Select Papers, G1.5.1. Inform Authors):

T6.1. SelectPapers

R7. Author (not connected to any goal as it is an interface to the user)

T7.1. WritePaper

T7.2. SubmitPaper

Communication Between Tasks:

C1. WritePaper → SubmitPaper

C2. SubmitPaper → CollectPapers : submit paper

C3. PartitionPapers → GetAbstracts : retrieve abstract

C4. PartitionPapers → AssignToReviewers : make assignments

C5. AssignToReviewers → NegotiatePapers : review papers

C6. NegotiatePapers → ReviewPaper

C7. ReviewPaper → DistributePapers : retrieve paper

C8. ReviewPaper → CollectReviews : submit review

C9. CollectReviews → SelectPapers

C10. SelectPapers → SubmitPaper : inform authors

3.4.5 Step 5. Concurrent Task Model

In this step, the LLM is prompted to generate a concurrent task model for each task in the role model, completing the analysis phase of the MaSE methodology.

Instructions:

For each Task defined in the Role Model, create a Concurrent Task Model represented as a Finite State Automaton (FSA). This model illustrates how the role performs its assigned task by transitioning through different states based on triggers (events), conditions (guards), and message transmissions (communication with other roles). This model details the role's behavior for task accomplishment; it may include interactions with other roles.

Important points:

1. *States Represent Internal Processing:*
    - *Each state in the model symbolizes a specific internal process or action performed by the role.*
    - *States may include activities in the format of:*
        - *data = action(parameters) → Represents data processing or logical operations.*
        - *action(parameters) → Represents a direct action or method call.*

2. Transitions Specify State Changes:

- *A transition defines how the role moves from one state to another and may include:*
  - *Trigger: The event that initiates the transition. It can be a message received from another role, expressed as: receive(message, sender_role_name)*
  - *Guard Condition: An optional condition that must be true for the transition to occur.*
  - *Transmission: Actions performed during the transition, such as sending a message to another role or triggering another task, expressed as: send(message, receiver_role_name)*

3. Start and End States:

- *Each task must have a Start and End state to mark the beginning and conclusion of the task's execution.*

4. Avoid Placeholders:

- *Use actual task names and role names as defined in the Role Model. Avoid placeholders.*

5. Communication Links:

- *All send and receive messages must reference real roles from the Role Model—no placeholder roles are allowed.*

Traceability Link:

Each Task in the Role Model should be detailed in a Concurrent Task Model. Therefore, the number of Concurrent Task Models should be equivalent to the number of tasks defined in the Role Model, ensuring full coverage and consistency. Please confirm that you generated a Concurrent Task Model for every task in the Role Model.

Output Template:

The model should follow the structured template below:

[Task Name – exactly match the task in the Role Model]

States:

S1. Start

S2. End

S3. [State Name] : data = action(parameter)

S4. [Next State Name]

Transitions:

T1. Start → [State Name] : receive([message], [sender_role_name])

T2. [State Name] → [Next State Name]

T3. [Next State Name] → End : [Cond]/send([message], [receiver_role_name])

Example:

For a Conference Management System, the task AssignToReviewers for Assigner role description is as follows:

"This task starts with Assigner role receiving a message with the papers from Partitioner role requesting makeAssigns, transitions from Start to MakeAssignments state. After performing internal processing for extracting the list of reviewers, the Assigner role transitions to RequestReviews state. After selecting a tuple of a reviewer and its assigned papers, the Assigner role transitions to Wait state by sending a message to the Reviewer role asking reviewPapers. The Assigner returns from Wait state to MakeAssignments state if the Reviewer declines and transitions to UpdatePaperList state if the Reviewer accepts. Finally, after removing the assigned papers from paper list, the Assigner role returns to the RequestReviews state if the paper list becomes empty or else transitions to end state by sending a message to Partitioner role saying the assignmentComplete."

Based on the above description, a Concurrent Task Model for this task may look like this:

AssignToReviewer

States:

S1. Start

S2. End

S3. MakeAssignments : list = getReviewers(papers)

S4. RequestReviews : <reviewer, paps>= removeTop (list)

S5. Wait

S6. UpdatePaperList : papers = update(papers, paps, reviewer)

Transitions:

T1. Start → MakeAssignments : receive(makeAssigns(papers), Partitioner)

T2. MakeAssignments → RequestReviews

T3. RequestReviews → Wait : /send(reviewPapers(paps), Reviewer)

T4. Wait → MakeAssignments : receive(decline(), Reviewer)

T5. Wait → UpdatePaperList : receive(accept(), Reviewer)

T6. UpdatePaperList → RequestReviews : [size(list) >0]

T7. UpdatePaperList → End : [size(list) = 0]/send(assignmentComplete(), Partitioner)"

3.4.6 Step 6. Agent Class Diagram

The first step of the design phase involves generating agent classes based on the roles. The LLM is prompted to create agent classes, assigning roles to each class.

Instructions:

Construct an Agent Class Diagram to represent agent classes and their associated roles, as defined in the Role Model. The diagram should depict:

1. *Agent Classes - Each agent class should encapsulate one or more roles, as defined in the Role Model.*
2. *Conversations - Communication should be specified between agents to represent message-based interactions.*

Important points:

1. *Agent Classes Represent Roles, Not Methods or Attributes:*
   - *Unlike traditional UML class diagrams, Agent Classes in MaSE contain Roles, not typical object-oriented attributes or methods.*
   - *Each agent class should explicitly list the roles it plays, exactly as defined in the Role Model.*
2. *Complete Role Assignment:*
   - *Every role in the Role Model must be assigned to at least one agent class.*
   - *If multiple roles are closely related (e.g., accessing the same data or performing sequential operations), consider grouping them in a single agent.*

3. Conversations Represent Communication:

- *The initiator is the agent that sends the initial message, and the responder is the agent that reacts.*
- *Each conversation should be labeled with the message name for clarity.*

4. Avoid Redundant or Passive Tasks:

- *All tasks, whether part of a conversation or internal, must represent meaningful, goal-driven activities.*
- *Avoid modeling passive tasks that merely receive, store, or forward data without decision-making, computation, or triggering a follow-up action.*
- *If a task performs no logic, merge it into a more meaningful task or refactor the role to better align with the system's goals.*

5. Avoid Placeholders:

- *Use real agent and role names as described in the Role Model—no placeholders.*

Traceability Link:

All roles from the Role Model must be represented in the Agent Class Diagram. Each agent class should fully account for its assigned roles, ensuring traceability to previous design steps. Please confirm that all roles in the Role Model assigned to a class in the Agent Class Diagram.

Output Template:

The model should follow the structured template below:

A1. [Agent Name] (playing: [List of Roles Plays - exactly match to the Roles in the Role Model])

A2. [Agent Name] (playing: [List of Roles Plays - exactly match to the Roles in the Role Model])

A3. And so on

Conversations:

C1. [Agent Name] → [Agent Name]: [Message Name]

C2. [Agent Name] → [Agent Name]: [Message Name]

C3. And so on

Example:

For a Conference Management System, we can define agent classes based on the roles:

Agent Classes:

A1. PCChair (playing: Partitioner, Collector, DecisionMaker)

A2. PCMember (playing: Assigner, Reviewer)

A3. DB (playing: PaperDB)

A4. Author (playing: Author)

Conversations:

C1. PCChair → PCMember: retrieve paper

C2. PCChair → DB: retrieve abstract

C3. PCChair → Author: inform authors

C4. PCMember → PCChair: make assignments

C5. PCMember → DB: collect reviews

C6. Author → DB: submit paper

3.4.7 Step 7. Communication Class Diagrams

In this step, the LLM is prompted to generate two communication class diagrams for each conversation between agents in the agent class diagram, one for the initiator agent and one for the responder agent.

Instructions:

For each Conversation between two agents defined in the Agent Class Diagram, generate two Communication Class Diagrams:

- *One for the Initiator Agent - represents the agent that initiates the conversation by sending a message to the responder as the first state transition.*
- *One for the Responder Agent - represents the agent that responds to the initiator's message and its state transition starts by receiving the initiator first message.*

These diagrams should be modeled as Finite State Automata (FSA), where:

- *States represent the agent's internal processing or waiting for a response.*

- *Transitions represent the communication (sending and receiving of messages) or internal actions that move the agent from one state to another.*
- *Within each Communication Class Diagram, all messages (send and receive) must occur only between the two agents involved in the conversation. Messages exchanged with any other agent should be modeled as a separate conversation with its own Communication Class Diagrams.*

Important points:

1. *Two Diagrams per Conversation: For every conversation defined, you need two diagrams:*
   - *One for the Initiator Agent.*
   - *One for the Responder Agent.*
2. *States and Transitions: Each diagram should have at least:*
   - *A Start state*
   - *One or more Processing states*
   - *An End state*
3. *Message Synchronization:*
   - *Any message that is sent by the initiator (^trans-mess) should be received by the responder (rec-mess).*
   - *This ensures that the communication is mirrored in both agents, reflecting real interactions.*
4. *Avoid Placeholders:*
   - *Use real conversation names and message names—no placeholders.*

Traceability Link:

All conversations between two agents listed in the Agent Class Diagram must be represented with two communication class diagrams (initiator and responder), ensuring full traceability and clarity of message exchange. Please confirm that you generated a pair of Communication Class Diagrams for each external conversation in the Agent Class Diagram, and that no other agent interactions are embedded within them.

Output Template:

The model should follow the structured template below:

[Conversation Name – exactly matches a conversation from Agent Class Diagram]

Initiator Agent: [Agent Name – exactly matches an agent from Agent Class Diagram]

States:

S1. Start

S2. [State Name]

S3. [State Name]

S4. End

Transitions:

T1. Start → [State Name]: rec-mess(args1) [cond] / action ^ trans-mess(args2)

T2. [State Name] → [State Name]: rec-mess(args1) [cond] / action ^ trans-mess(args2)

T3. [State Name] → End: rec-mess(args1) [cond] / action ^ trans-mess(args2)

Responder Agent: [Agent Name – exactly matches an agent from Agent Class Diagram]

States:

S1. Start

S2. [State Name]

S3. [State Name]

S4. End

Transitions:

T1. Start → [State Name]: rec-mess(args1) [cond] / action ^ trans-mess(args2)

T2. [State Name] → [State Name]: rec-mess(args1) [cond] / action ^ trans-mess(args2)

T3. [State Name] → End: rec-mess(args1) [cond] / action ^ trans-mess(args2)

Example:

For a Conference Management System, consider a sample conversation from the Role Model named SubmitPaper initiated by UserInterfaceAgent and responded by PaperManager.

inform authors

Initiator Agent: PCChair

States:

S1. Start

S2. Wait

S3. Update : updatePapers(paper)

S4. End

Transitions:

T1. Start → Wait : ˆnotice (accept, paper)

T2. Wait → Update : decline()

T3. Wait → End : accept()

T4. Update → End

Responder Agent: Author

States:

S1. Start

S2. RecheckAvailability : avail = checkAvail(paper)

S3. End

Transitions:

T1. Start → RecheckAvailability : notice(accept, paper)

T2. RecheckAvailability → End : [avail] ˆ accept()

T3. RecheckAvailability → End : [NOT avail] ˆ decline()

3.4.8 Step 8. Agent Architecture

This step involves generating agents' internal architecture. The LLM is prompted to generate class diagrams showing the internal architecture of each agent.

Instructions:

For each Agent defined in the Agent Class Diagram, generate a Class Diagram representing its internal architecture. This includes:

1. *Internal Classes - The components that make up the agent's structure.*
2. *Attributes - The internal state variables or properties of each class.*
3. *Operations - Methods or actions that correspond to activities in the Communication Class Diagrams from the previous step.*

4. *Communications - Any internal communication paths between the agent's internal classes*

Important points:

1. *One Diagram per Agent:*
    - *Each agent from the Agent Class Diagram should have its own internal architecture diagram.*
2. *Mapping Activities to Operations:*
    - *All activities identified in the Communication Class Diagrams should be represented as operations in the internal classes of the agent.*
3. *Internal Classes:*
    - *The agent should be decomposed into logical classes that reflect its roles and activities.*
4. *Attributes Representation:*
    - *Any internal state or configuration that affects operations should be represented as attributes.*
5. *Communications Between Internal Classes:*
    - *If one internal class interacts with another, this should be represented with an arrow, indicating the flow of data or method calls.*
6. *Avoid Placeholders:*
    - *Use real class names, attributes, and operations—no placeholders.*

Traceability Link:

All agents in the Agent Class Diagram should be fully represented, and all activities from the Communication Class Diagrams must be mapped as operations in the internal classes of the respective agents. Please confirm that you generated an architectural model for each agent in the Agent Class Diagram.

Output Template:

The model should follow the structured template below:

[Agent Name – exactly matches an agent from Agent Class Diagram] Internal Architecture

Internal Classes:

C1. [Class Name]

- Attributes:

- [Attribute Name]: [Type]

- [Attribute Name]: [Type]

- And so on...

- Operations:

- [Operation Name]([Parameters])

- [Operation Name]([Parameters])

- And so on...

C2. [Class Name]

- Attributes:

- [Attribute Name]: [Type]

- [Attribute Name]: [Type]

- And so on...

- Operations:

- [Operation Name]([Parameters])

- [Operation Name]([Parameters])

- And so on...

Communications:

C1. [Class Name] → [Class Name] (if there is any relationship between classes)

C2. And so on....

Example:

PCChair:

C1. Partitioner

- attributes:

- papers: List

- Operations:

- abstractReceived(paper)

- partitionPapers(papers)

C2. Collector

- attributes:

- papers: List

- reviews: List

- Operations:

- collectReviews(paper)

C3. DecisionMaker

- attributes:

- papers: List

- reviews: List

- Operations:

- selectPapers(papers, reviews)

Communications:

C1. Partitioner → Collector

C2. Collector → DecisionMaker

3.4.9 Step 9. Deployment Diagram

In the final step of the MaSE design phase, the LLM is prompted to generate a deployment diagram, detailing how agent instances will be deployed in a distributed system.

Instructions:

Construct a Deployment Diagram to represent the physical or logical distribution of agents within the system. This diagram should display:

1. *Agent Instances - Each agent class instantiated in the system, with its role clearly defined. All agents must have at least one instance in the diagram.*
2. *Computational Nodes - The physical or virtual locations (servers, databases, user interfaces, etc.) where agents are deployed.*
3. *Communication Links - The interactions and message exchanges between agents, mapped to network or logical communication paths.*

Important points:

1. *Agent Instances and Roles:*
    - *Every agent class defined in the Agent Class Diagram must be instantiated in the Deployment Diagram.*
2. *Computational Nodes Representation:*
    - *Each agent must be mapped to a computational node representing where it operates.*
    - *Examples of nodes:*
        - *WebServer for interface agents.*
        - *DatabaseServer for data access agents.*
3. *Communication Paths:*
    - *Draw lines between agent instances to represent message exchanges.*
    - *Ensure these paths reflect the conversations defined in the Agent Class Diagram.*
4. *Group Agents Logically or Physically:*
    - *Agents that communicate frequently or are part of a common subsystem should be grouped logically or on the same computational node.*
    - *Consider load balancing and redundancy for critical agents.*
5. *Avoid Placeholders:*
    - *Use real agent names, node names, and communication labels—no placeholders.*

Traceability Link:

The deployment of each agent instance should directly correspond to the Agent Class Diagram and the Communication Class Diagrams. Please confirm that your generated Deployment Diagram contains one or more instances of all agents in the Agent Class Diagram.

Output Template:

The model should follow the structured template below:

Computational Nodes:

N1. [Node Name]

- [AgentInstanceName : AgentClassName]

- [AgentInstanceName : AgentClassName]

N2. [Node Name]

- [AgentInstanceName : AgentClassName]

- [AgentInstanceName : AgentClassName]

Communication Paths:

C1. [AgentInstanceName] → [AgentInstanceName] : [Message/Event Name]

C2. [AgentInstanceName] → [AgentInstanceName] : [Message/Event Name]

C3. And so on...

Example:

For a Conference Management System, the deployment diagram might look like this:

Computational Nodes:

N1. Node1

- Chair : PCChair

N2. Node2

- DB: Database

N3. Node3

- A1 : Author

- PCM1 : PCMember

N4. Node4

- A2 : Author

N5. Node5

- A3 : Author

Communication Paths:

C1. Chair – DB

C2. Chair – PCM1

C3. Chair – A1

C4. Chair – A2

C5. Chair – A3

C6. DB – PCM1

C7. DB – A1

C8. DB – A2

C9. DB – A3

3.4.10 Step 10. Code

The final step of the process is generating the implementation code. To manage the complexity of this task, we divided it to five sub-steps.

Ontology Classes and Schemas:

First, the LLM is prompted to generate ontology classes, concepts, and their corresponding schemas. These definitions provide the structure for messages' content.

Instructions:

In the following step, I want you to generate JADE code for designed MAS system in five sub-steps. For the first sub-step, please generate JADE code for required ontology classes, also define and register concepts and their schemas as needed in the system. Remind to make correct inheritance from JADE ontology classes. The code should follow the official method signatures and conventions provided in the official JADE tutorial. These generated classes will be used in Agent classes for data exchange.

Important points:

1. *Create concepts, schemas and add them to ontology, so the concepts can be used in message interactions between agent classes.*
2. *Define ontology concepts that directly reflect message content from the Communication Class Diagrams (e.g., Paper, ReviewRequest, DecisionNotice). Use the Goal Hierarchy and Role Model to determine which entities need structured message exchange.*
3. *Don't delegate any implementation or direction to me to complete your generated code. Your generated code must be complete for the requested sub-step.*
4. *The code should be syntactically correct by following the official JADE tutorial.*
5. *The code should be complete and ready to run.*

6. *Please consider a separated file for each class.*

7. *The code should be well-documented and easy to understand.*

8. *The code should be error-free and handle all possible exceptions.*

Traceability Links:

The code should be generated considering the structural and behavioural models generated in previous steps. Please confirm that you generated code for all required concepts, developed concept's schemas and added them to ontology classes based on the system designed models.

Output Template:

The JADE code for this part of the system. The target platform is jade-4.6.0. The code should be in Java and follow the JADE platform conventions.

Agent Class Implementation:

In the second sub-step, the LLM is prompted to generate code for the agent classes. This code should be based on the previously designed agent architecture and must reflect the agents' communication behaviour. Where applicable, we instruct the LLM to assume a placeholder user interface class (e.g., Agent_UI) for agents requiring human interaction.

Instructions:

For the second sub-step, please generate JADE code for the agent classes in the system using the defined ontology from the previous step. Implement each agent class based on the designed Agent Architecture, including internal classes and their attributes and operations. Based on the Communication Class Diagram for agent's external message passing, define each agent's behaviors as send or receive messages.

For Agents that need a user interface class for human-interaction. Assume they already have a user interface class names agent_UI, instantiate their user interface at agent initialization and implement the interactions with the interface as needed; these UI classes will be implemented later.

Important points:

1. *Add Console logs in the code for every new action in agent like setup, take down, starting a behaviour or sending and receiving a message.*
2. *Don't use hardcoded agent names. use dynamic registration and discovery using JADE's Directory Facilitator (DF).*
3. *Don't use string-based messages for agent communications. Instead use previously defined ontology to define message's structure.*
4. *Don't delegate any implementation or direction to me to complete your generated code. Your generated code must be complete for the requested sub-step.*
5. *The code should be syntactically correct by following the official JADE tutorial.*
6. *The code should be complete and ready to run.*
7. *Please consider a separated file for each class.*
8. *The code should be well-documented and easy to understand.*
9. *The code should be error-free and handle all possible exceptions.*

Traceability Links:

The code should be totally aligned with Agent Classes in the Agent Class Diagram. Their internal implementation should be aligned with Agent Architecture diagram, and their behavior should be implemented as Concurrent Task Model and Communication Class Diagram. Please confirm that you generated code for all agents according to their designed internal architecture and behavior.

Output Template:

The JADE code for this part of the system. The target platform is jade-4.6.0. The code should be in Java and follow the JADE platform conventions."

Containers Implementation:

The third sub-step focuses on generating code for agent containers based on the deployment diagram. The LLM is asked to instantiate at least one instance of each agent type in the system. Additionally, it is instructed to include code for simulating the system's main scenario by sending a test message to the

initiator agent.

Instructions:

For the third sub-step, please generate JADE code for the container classes in the system. Don't forget to enable RMA in the MainContainer. The containers should align with Nodes in the Deployment Diagram, and they have to instantiate their agent instances.
Use the previously implemented ontology and agent classes. All agent classes should have at least one instance according. After instantiating all agent classes, add a test code for the system main scenario testing. This is done by sending a message to the scenario's initiator agent class. The interaction between the agents for the scenario realization should be visible in the JADE sniffer tool.

Important points:

1. *Don't delegate any implementation or direction to me to complete your generated code. Your generated code must be complete for the requested sub-step.*
2. *The code should be complete and ready to run.*
3. *Please consider a separated file for each class.*
4. *The code should be well-documented and easy to understand.*
5. *The code should be error-free and handle all possible exceptions.*

Traceability Links:

The code should be generated considering the Deployment Diagram. The code should be compatible with previously generated codes. Please confirm that you generated code for containers and instantiate all agents according to Deployment Diagram.

Output Template:

The JADE code for this part of the system. The target platform is jade-4.6.0. The code should be in Java and follow the JADE platform conventions.

User Interface Implementation:

In this sub-step, the LLM is prompted to generate simple java-based form interfaces for agents that interact with human users. These UI classes should be initialized and terminated alongside their corresponding agents.

Instructions:

For the fourth sub-step, please implement the user interfaces for user interface class assumed in the second step using simple java forms for better interactivity and completeness. They should be already connected to their agent and initialized with agent initialization and interact with their agent as required.

Important points:

1. *Don't delegate any implementation or direction to me to complete your generated code. Your generated code must be complete for the requested sub-step.*
2. *The code should be complete and ready to run.*
3. *Please consider a separated file for each class.*
4. *The code should be well-documented and easy to understand.*
5. *The code should be error-free and handle all possible exceptions.*

Traceability Links:

The code should be compatible with previously generated codes. Please confirm that you generated code for all user interface classes named agent_UI and linked them to their agents.

Output Template:

The JADE code for this part of the system. The target platform is jade-4.6.0. The code should be in Java and follow the JADE platform conventions.

Code Review and Final Corrections:

Finally, the LLM is asked to review the entire codebase, make any necessary correction, resolve inconsistencies, and ensure all components are properly integrated. Only modified or newly added code should be returned in this step.

Instructions:

Finally, please review your generated code in the previous four steps as a whole MAS and make correction if needed. Just give code for newly added classes or edited classes.

Important points:

1. *Don't regenerate all codes, just generate code for new classes or edited classes.*
2. *Don't delegate any implementation or direction to me to complete your generated code. Your generated code must be complete for the requested sub-step.*

Traceability Links:

The code should be compatible with previously generated codes.

Output Template:

The JADE code for this part of the system. The target platform is jade-4.6.0. The code should be in Java and follow the JADE platform conventions.

The outputs generated during this process, comprising analysis and design artefacts as well as implementation code, serve multiple purposes. Beyond providing valuable resources for system testing and maintenance, these artefacts enable the LLM to maintain traceability links to earlier outputs, ensuring alignment with project requirements and goals. This traceability ensures that the implementation code remains consistent with the defined system requirements and the overall system design.

By integrating these structured steps and leveraging LLM-generated outputs, the methodology achieves a seamless transition from system requirements to implementation while preserving alignment and consistency throughout the development process.

3.5 LLM-based MaSE developed application

An LLM-based MaSE application was developed based on the proposed framework. The application provides a user-friendly interface for users to input the system name and requirements, process them through the methodology, follow the development process steps, view the corresponding prompts and results (including visual representation if applicable), download the results, and navigate between steps. The list of features developed in the application are as follows:

1. Users can input the system name and requirements.
2. Users can follow the development process step-by-step.

3. Users can view and modify the current step's prompt as needed.

4. Users can view and modify the results of the current step as needed.

5. Users can send feedback to the LLM and request refinements.

6. Users can download the results.

7. Users can navigate between steps and request to rerun steps.

From the above features, we chose to focus on core functionality for the first version, with the ability to modify prompts and results, and provide feedback, to be implemented in future versions. The following sections provide details on the application's structure and architecture.

3.5.1 Technical Stack

The application was developed as a web application using python, utilizing the flask framework for the backend and React for the frontend. Flask was selected for its lightweight architecture, which is suitable for the application's simple API needs.

3.5.2 Architecture and Design

The application follows a client-server architecture with a RESTful API for communication between React frontend and the Flask backend. The backend handles business logic, while the frontend is responsible for presenting data to the user. As demonstrated in Figure 3.2, the system consists of the following main parts:

1. **Mediator:** This component serves as the interface to the system's internal parts, receiving requests and sending them to the appropriate components.

2. **MaseTracker:** This component initializes the MaSE process, manages the current step, provides prompts for each step, and saves the resulting outputs.

3. **LlmConnector:** This component interfaces with the LLM API for communication.

4. **PlantumlConvertor:** This component convers textual model results into PlantUML code for visualization in the browser, enhancing readability and understandability.

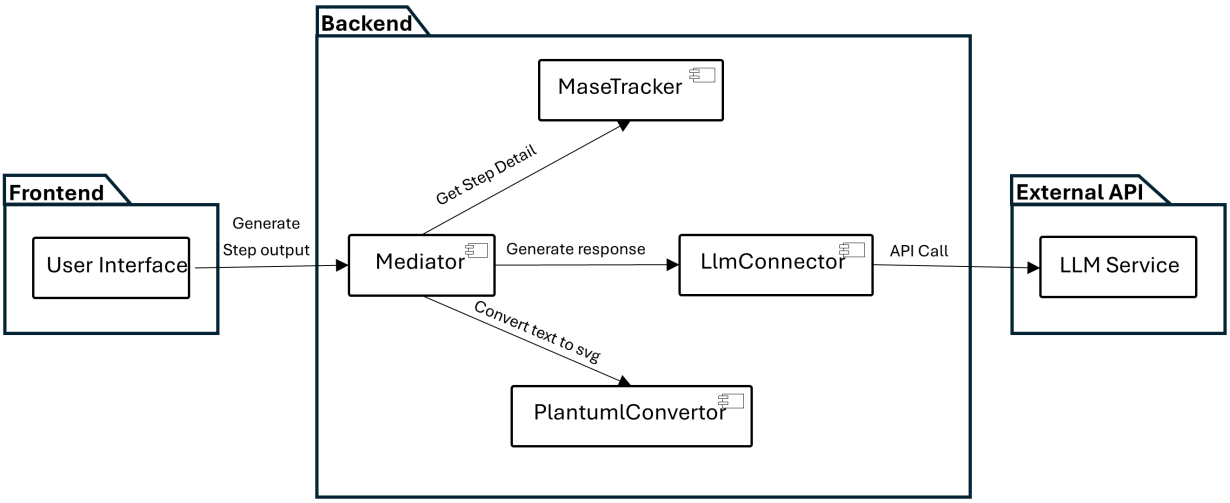

Figure 3.2: LLM-based MaSE Application's Architecture

3.5.3 User Interface and User Experience

The application features a minimalist interface with a workflow bar on the left, showing the development process and the current step (see Figure 3.3). The main area displays the prompt, result text and result models, with options to navigate to the next or previous step. Users can ask for models or code generation at each step and download the results. Figure 3.4 provides a screenshot of the application's main view.

3.5.4 Application Workflow

Upon launching the application, users are prompted to enter the system name and requirements in the starting form. The system then processes this information and displays it in a triple-tab format: prompt, result, and model. Users can request model generation, navigating to the next step, and download the results in the provided format.

3.5.5 Challenges and limitations

The application currently does not allow users to modify prompts or outputs, nor it incorporate user feedback into generated results. Initially, the goal was to develop a simple version of the application and then add more features in subsequent releases. Additionally, user feedback in the form of new prompts in the middle of the development process could sometimes divert the LLM from its intended path, so we plan to introduce feedback cautiously, starting with a predefined list of expert user feedback options for refinement.

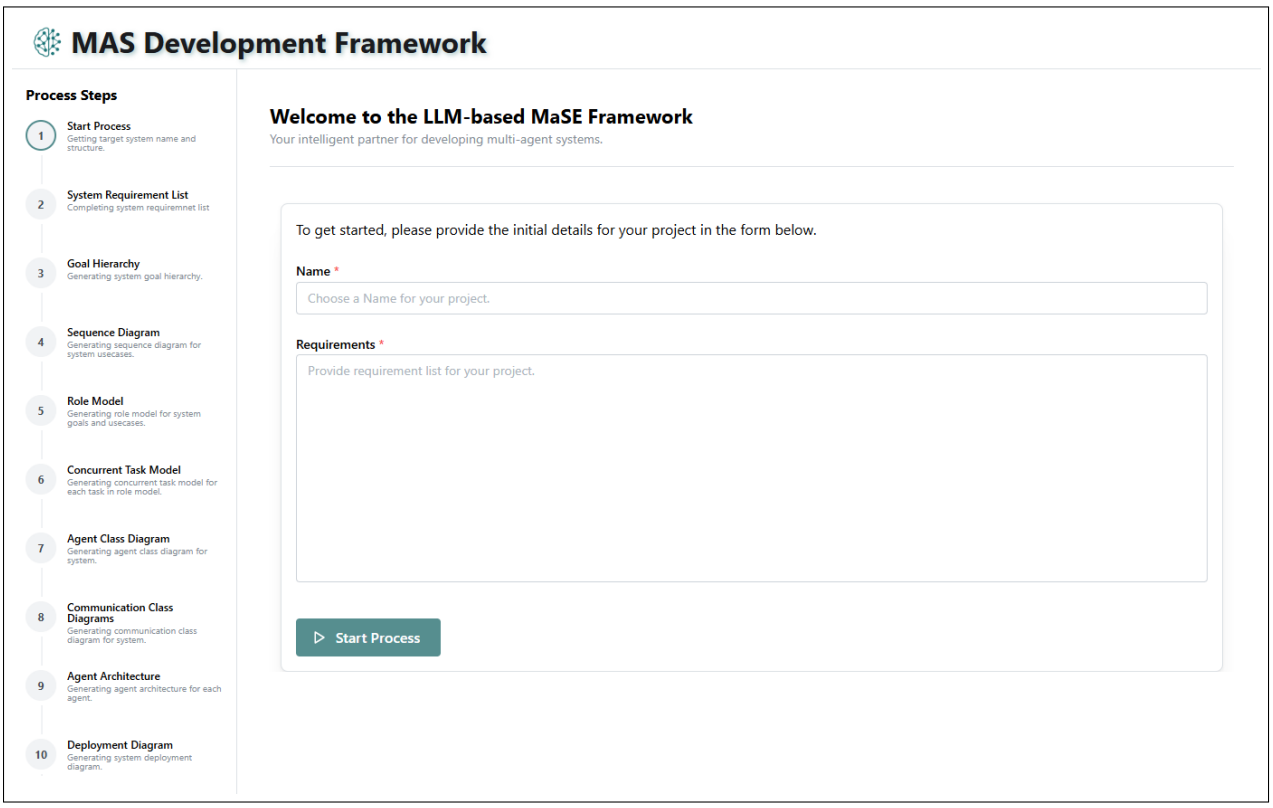

Figure 3.3: The Start View of the LLM-based MaSE Application

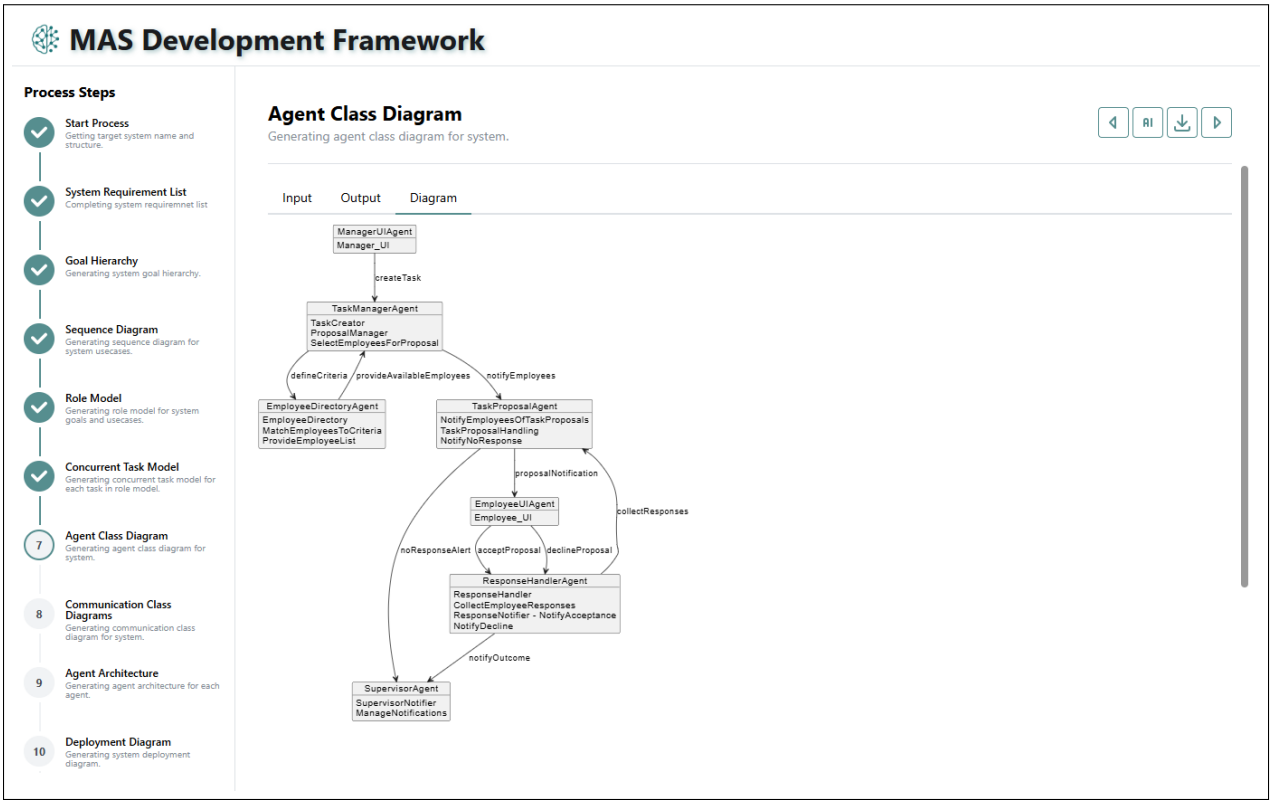

Figure 3.4: LLM-based MaSE Application Screenshot

Chapter 4

Evaluation - Case Study

4.1 Introduction

The evaluation of Large Language Models (LLMs) remains a challenging and underexplored area. Assessing the performance of these models requires a tailored evaluation methodology [41]. As stated in [31], one of the primary challenges in exploring LLM for domain-specific software engineering tasks, such as multi-agent system (MAS) development, is the lack of domain-specific datasets and ground truth. In the case of our proposed framework, no existing benchmarks are available, and due to its focus on agent-oriented development, there are no directly comparable works. As a result, we adopted alternative evaluation methods, including empirical software engineering. Empirical studies, which have gained increasing attention in recent years, offer a scientific approach to evaluation. In [42], detailed approaches and guidelines for conducting experiments and empirically assessing techniques, tools, and methods in software engineering are provided. We leveraged these insights in our evaluation process.

Case studies and experiments are widely used approaches in empirical evaluation. A case study involves investigating a specific instance of a software engineering phenomenon within its real-life context, especially when the distinction between the phenomenon and its context is unclear. This method is particularly suitable for industrial evaluations of methods or tools. However, the results of case studies are often difficult to interpret or generalize, which means they typically do not provide broad statistical conclusions [42].

In contrast, an experiment or controlled experiment focuses on examining the impact of manipulating a single factor or variable on outcome variables. This approach is ideal for exploring relationships and comparing alternatives. Because experiments allow for greater control over factors such as subjects and objects, more generalizable conclusions can be drawn. Additionally, statistical analysis based on hypothesis

testing methods can be performed, providing more robust and reliable insights [42].

We organized the evaluation process in two phases. The first phase involved a case study, conducted outside an industrial environment, with the goal of performing an exploratory assessment and qualitatively comparing human-generated models and code with those produced by large language models (LLMs). This phase aims to document observations and extract a list of hypotheses. In the second phase, a key hypothesis was selected for further investigation, and an experiment was conducted to evaluate it quantitatively and analyze it statistically.

This chapter provides a detailed description of the case study, while the next chapter focuses on the experimental procedure.

4.2 Exploratory Application and Qualitative Evaluation

This chapter presents the application of our LLM-based MaSE framework on a sample project. It includes detailed descriptions of the process, observations, and a qualitative assessment. This analysis, along with the observations, leads to the formulation of hypotheses regarding the framework's impact and effectiveness.

4.3 Preparations

The framework evaluation required two key components: an LLM and some multi-agent projects. For the LLM, we selected the GPT model, which is widely considered as the best model with conversational prompting [8]. As a decoder-only architecture, GPT excels at generating tasks, making it well-suited for our framework's requirements. Additionally, we gathered a collection of university projects from the courses related to MAS development. These projects, developed by anonymous graduate students, followed the MaSE methodology and were implemented using the JADE platform.

Using GPT and this curated group of projects allowed us to conduct our framework's empirical evaluations.

4.4 Case Study

We tested the LLM-based MaSE framework on a sample project, comparing human-generated and LLM-generated outputs. This phase aimed to document observations and to develop hypotheses for future experiments. The sample project, a "Resource Management System", includes the following requirements:

The basic requirements of our system are the ability to provide the manager the possibility to create a task and provide him/her with the list of employees able to perform the task specified based on certain criteria defined by the manager. The system must allow the supervisor to select an employee from the proposed list and send him/her a proposal to perform the task. This latter should be able to accept or decline the offer within a certain deadline. More specifically, the system aims at:

1. *Create a task*
2. *Request availability from the list of available employees*
3. *Select the list of employees that matches the criteria*
4. *Submit offer to employees*
5. *The employees can accept or decline the offer*
6. *Collect acceptance or refusal of the offer*

For each step of the methodology, we:

1. Provided a brief description.
2. Presented both the human and LLM-generated results produced from the same system requirements.
3. Evaluated the outputs using qualitative criteria such as structure, completeness, traceability, and adherence to MaSE.

4.4.1 Requirement List

In the first step, we tasked the LLM with refining the requirement list. We believe that this polished list, enables the LLM to maintain alignment with requirements throughout the process and facilitates traceability analysis between generated artefacts and defined requirements. The LLM response for this step is a list of polished requirements as follows:

R1. The system must provide functionality for the manager to create a new task, including specifying task details such as title, description, deadline, and any specific employee criteria.

R2. The system must allow managers to define criteria for employee selection, such as skills, experience, or certifications, to filter employees suitable for a task.

R3. Upon creating a task, the system should compile and output a list of employees available to perform the task according to the manager-defined criteria.

R4. The system must allow the supervisor to review and select employees from the compiled list who can be offered the task.

R5. The system should enable supervisors to send a task offer to selected employees, including task details and the acceptance deadline.

R6. Employees must have the ability to receive task offers and respond by accepting or declining the task within the specified deadline.

R7. The system should automatically collect and record the responses from employees, noting which offers have been accepted or declined.

R8. The system should provide a summary of task allocations and response statuses to the manager for task tracking and reporting.

R9. In the event an employee declines an offer, the system must notify the supervisor and provide options for reassigning the task to other available employees.

R10. The system must ensure that all communications related to task offers and responses are logged and traceable for future reference.

4.4.2 Goal Hierarchy

This step involved both human and LLM-generated goal hierarchies based on the system requirements. Figure 4.1 and Figure 4.2 show the output for human and LLM, respectively.

We utilized the traceability to requirements presented in Table 4.1 for making the comparison. The list of requirements was extracted from system initial requirements. While both sets of goals cover the main functionality of the system, the human-generated model in some cases did not number the sub-goals correctly and some goal names are vague. On the other hand, the LLM-generated model is well designed in terms of goal-subgoal relationships and goals' names.

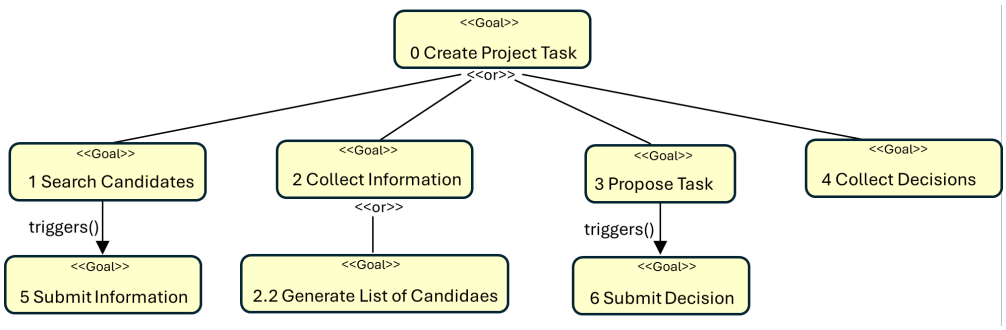

Figure 4.1: Human-generated Goal Hierarchy for RMS

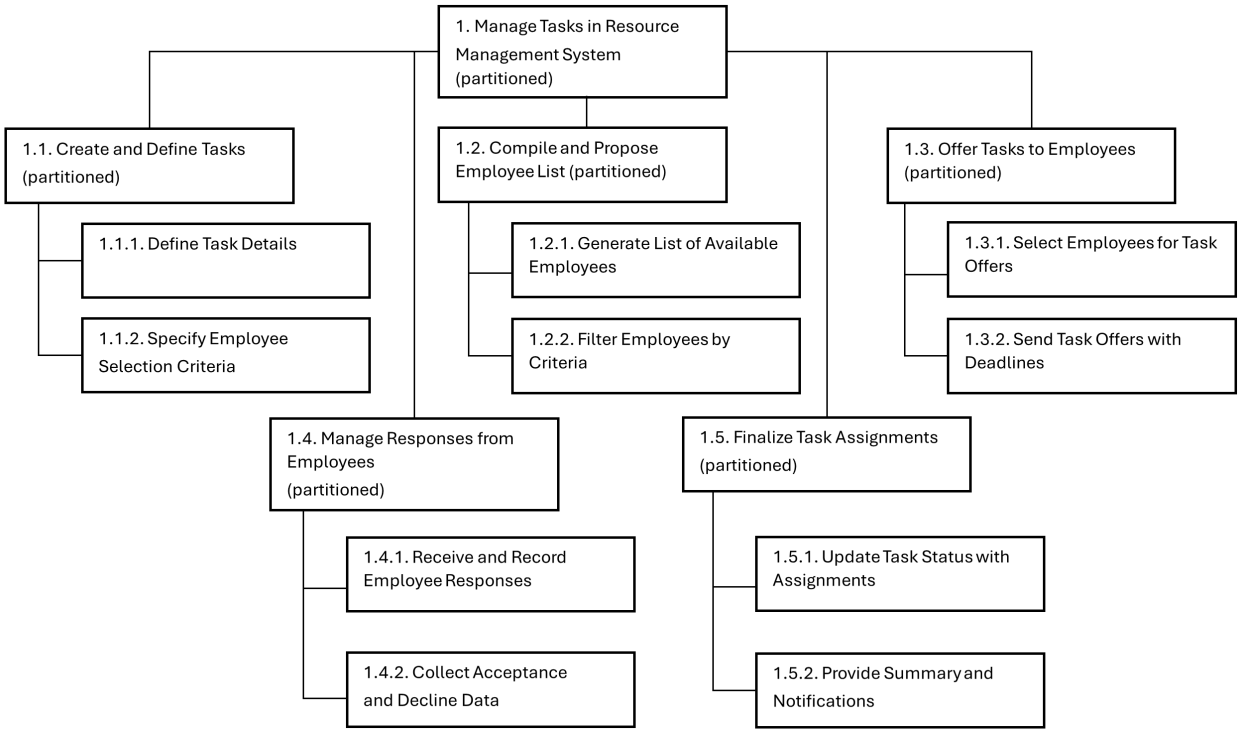

Figure 4.2: LLM-generated Goal-Hierarchy for RMS (Manually Visualized)

4.4.3 Use cases scenarios

The third step of the methodology is to extract user scenarios from the system requirements and model them as sequence diagrams. The human-generated sequence diagram is shown in Figure 4.3 and the LLM-generated response was manually visualized and illustrates in Figure 4.4.

All system functionality is demonstrated in one high-level use case in human-generated sequence diagram. While LLM modeled the system functionality through six use cases scenarios. Based on the generated sequence diagrams and the traceability displayed in Table 4.2, the LLM generated more complete and finer-grained set of sequence diagrams then the other one generated by human.

#	Requirement	Human-generated goal-hierarchy item	LLM-generated goal-hierarchy item
1	Allow the manager to create a task	0	1.1.1
2	Allow the manager to define criteria for employee selection	2	1.1.2
3	Get list of eligible employees	2.2	1.2.1, 1.2.2
4	Enable supervisor to select employee	1, 5	1.3.1
5	Send task proposal	3	1.3.2
6	Allow the employee to accept or decline the offer	6	1.4.1
7	Collect and process responses	4	1.4.2
8	Set a deadline for response	–	1.3.2

Table 4.1: Goal Hierarchy trace back to Requirements.

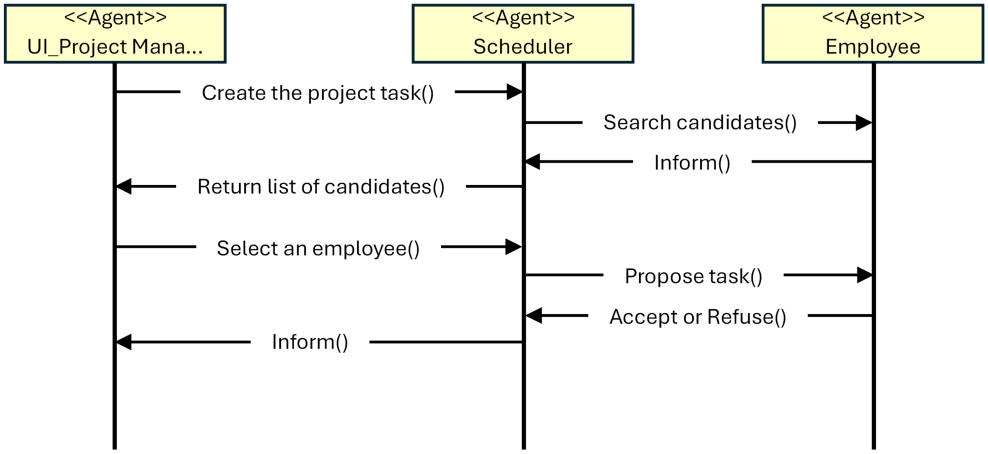

Figure 4.3: Human-generated Sequence Diagram(s) for RMS

4.4.4 Role model

In the role model, roles and tasks are extracted based on system goals and scenarios. The human-generated role model is as shown in Figure 4.5, and the LLM-generated results are manually visualized in Figure 4.6.

Human-generated role model contains roles with general names such as “Creator” and “Distributor” and relationship between tasks in not displayed. While the LLM-generated role model contains roles with more specific names and the relationship between the tasks are included. There is a one-to-one relationship between goals in the goal hierarchy and the tasks in the role model generated by human, but the LLM output converted the goals to more fine-grained tasks. traceability links to goals are depicted in Table 4.3.

4.4.5 Concurrent task model

For each task in the role model, a concurrent task model should be generated to detail the behavior for that task. Figure 4.7 and Figure 4.8 display the human and LLM-generated concurrent task models.

Only one model is generated by human that does not cover the tasks behaviors completely. Also, the

Figure 4.4: LLM-generated Sequence Diagram(s) for RMS (Manually Visualized)

name of roles is not compatible with the ones defined in the role model. Instead, LLM generated a model for each task in the role model and the roles name is completely compatible with the roles defined in the role model.

#	Requirement	Human-generated Sequence Diagram	LLM-generated Sequence Diagram
1	Allow the manager to create a task	✓	✓
2	Allow the manager to define criteria for employee selection	✗	✓
3	Get list of eligible employees	✓	✓
4	Enable supervisor to select employee	✓	✓
5	Send task proposal	✓	✓
6	Allow the employee to accept or decline the offer	✓	✓
7	Collect and process responses	✓	✓
8	Set a deadline for response	✗	✓

Table 4.2: Sequence Diagram(s) trace back to Requirements.

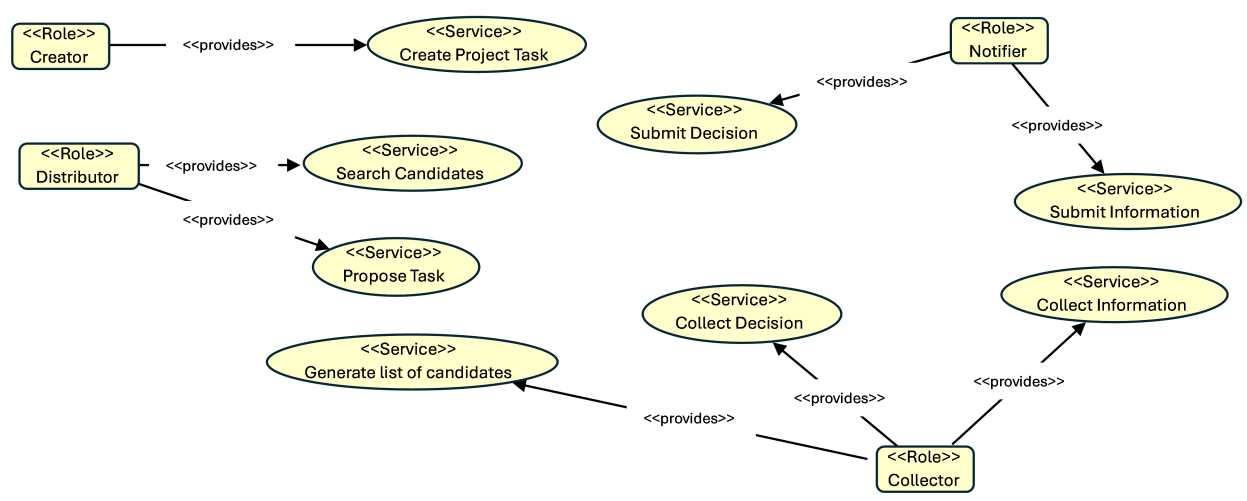

Figure 4.5: Human-generated Role Model for RMS

4.4.6 Agent class diagram

This step involves mapping roles to agent classes and determining which role will be played by which agent at design level. An agent will be playing one or more roles in the system, and the communication between roles that assigned to different agents will be the conversation between agents, specified with an arrow from the initiator to the responder agent. Figure 4.9 is the agent class diagram generated by the human. The LLM output is manually visualized in Figure 4.10.

In human-generated Agent Class Diagram, two agent classes are playing 4 roles with two general communications named “Query” and “Inform” and, the LLM distributed 8 roles to 4 agents with 5 communications. While both models assigned all roles to agent classes, the LLM defined more specific communications between the agents.

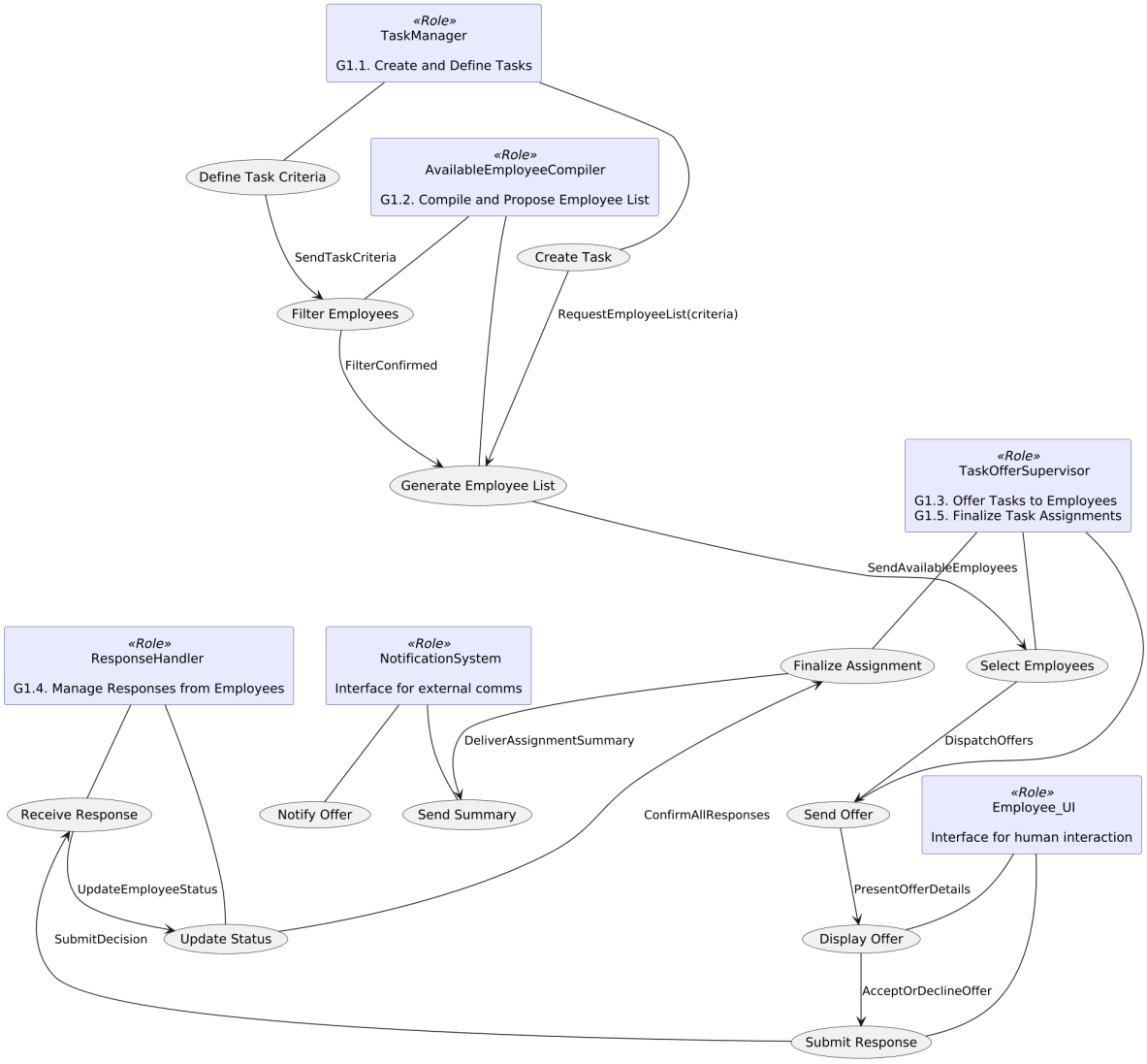

Figure 4.6: LLM-generated Role Model for RMS (Manually visualized)

4.4.7 Communication class diagram

For each communication between agents in the class diagram, two communication class diagrams should be developed, one for the initiator agent and one for the responder agent. Figure 4.11 displays the human-generated models, and the LLM output is manually visualized in Figure 4.12.

Although the communication between agents was defined very generally in the agent class diagram generated by human, the created communication class diagram does not reflect those general relationships and is limited to a specific communication between agents namely “Search Candidates”. On the other hand, LLM generated a pair of models for all 5 communications in the agent class diagram that display the

Human-generated Goals	Human-generated Roles	LLM-generated Goals	LLM-generated Roles
0	Creator	1.1 (1.1.1, 1.1.2)	TaskManager
1	Distributor	1.2 (1.2.1, 1.2.2)	AvailableEmployeeCompiler
2	Collector	1.3 (1.3.1, 1.3.2)	TaskOfferSupervisor
2.2	Collector	1.4 (1.4.1, 1.4.2)	ResponseHandler
3	Distributor	1.5 (1.5.1, 1.5.2)	TaskOfferSupervisor
4	Collector		
5	Notifier		
6	Notifier		

Table 4.3: Traceability from Roles to Goals.

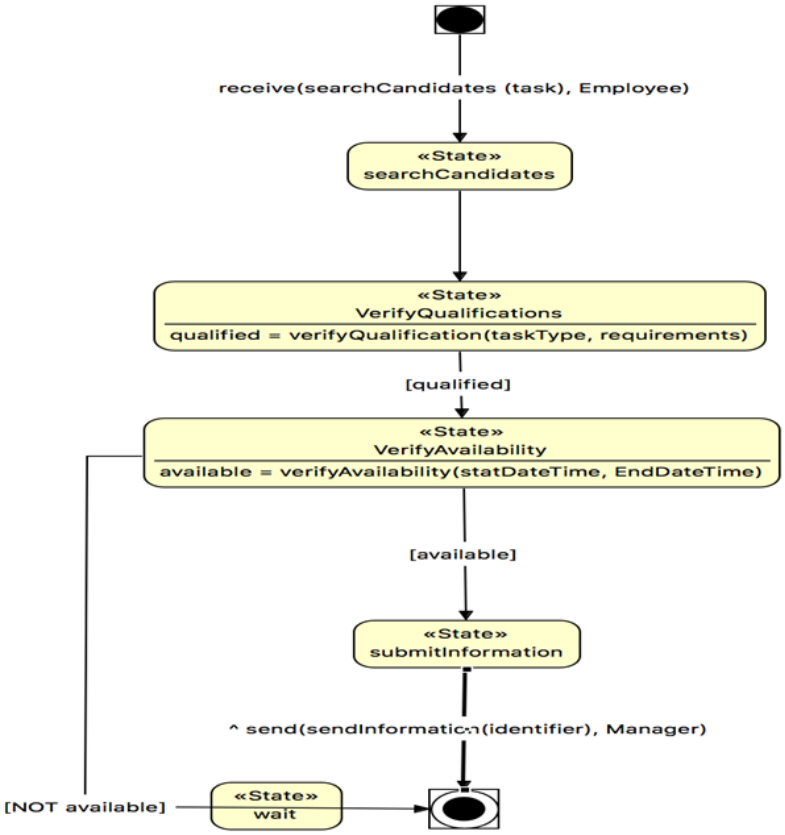

Figure 4.7: Human-generated Concurrent Task model for RMS

communication between the involved agents correctly.

4.4.8 Agent Architecture

In this step the internal design of each agent should be modeled in a class diagram. Human did not generate any model for this step, while the LLM detailed all agents' internal architecture in a model displayed in Figure 4.13.

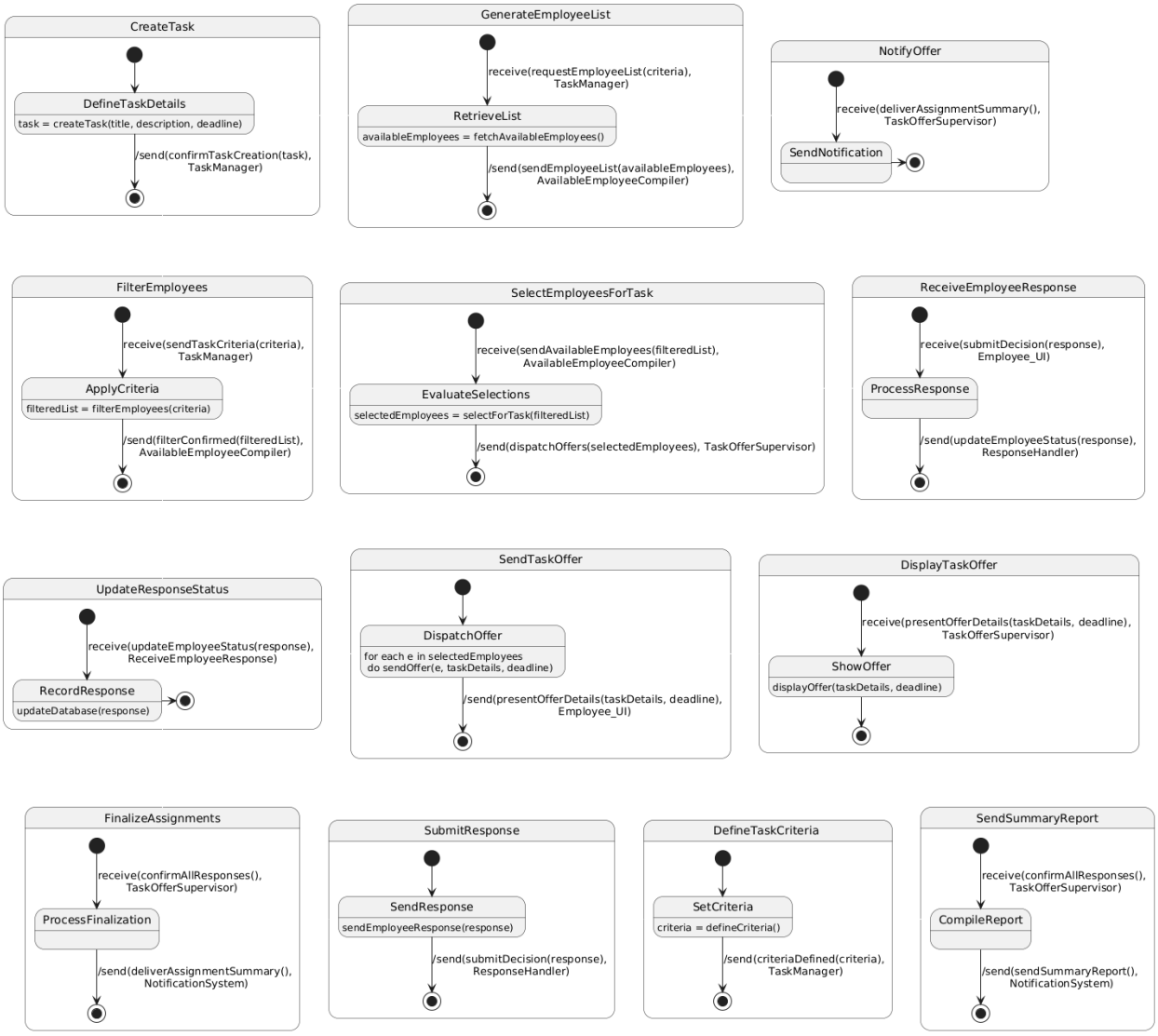

Figure 4.8: LLM-generated Concurrent Task Models for RMS

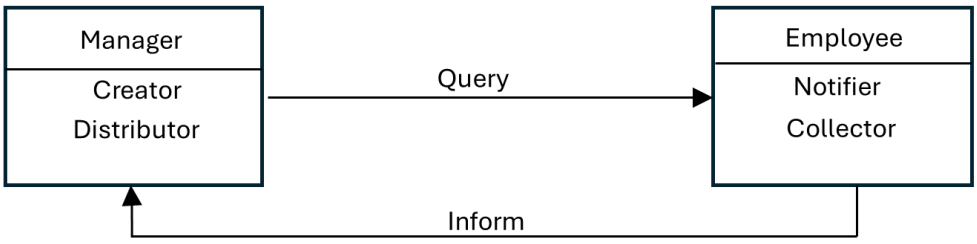

Figure 4.9: Human-generated Agent Class Diagram for RMS

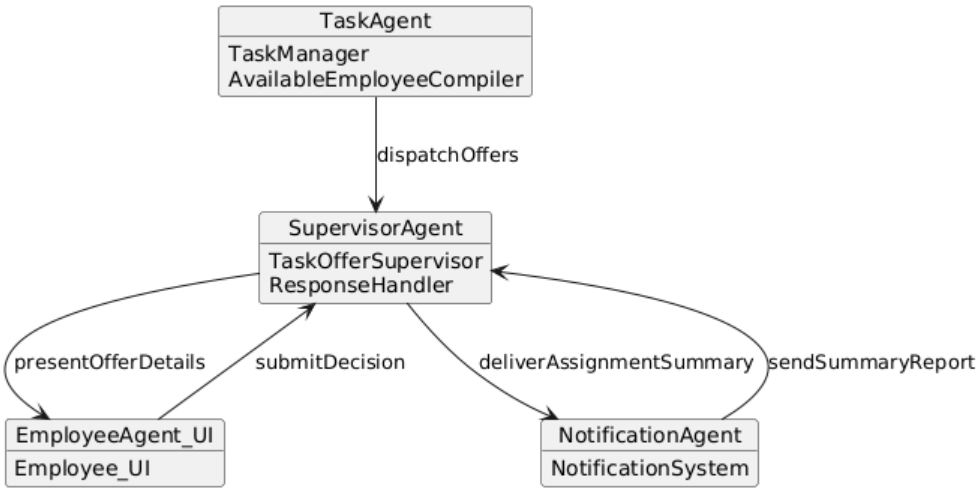

Figure 4.10: LLM-generated Agent Class Diagram for RMS

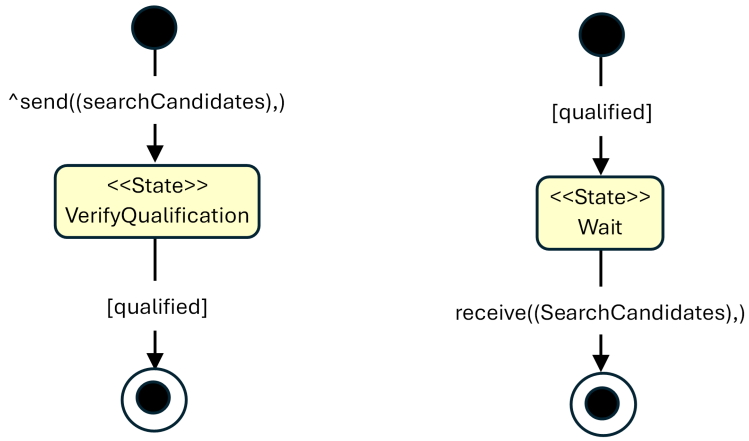

Figure 4.11: Human-generated Communication Class Diagrams for RMS

4.4.9 Deployment Diagram

The last step of the design phase involves generating a deployment diagram for the system. Figure 4.14 and Figure 4.15 demonstrates the human and the LLM-generated deployment diagram respectively. Both models include at least one instance of every agent and displayed the relationship between the instances. While the human generated a simple model, the LLM created a more detailed diagram including more information about the containers and the relationship between the agent instances.

4.4.10 Code

The final step of the process is the implementation of the system. Both human and LLM-generated codes largely adhered to their respective design documents. Agents were implemented based on Agent Class

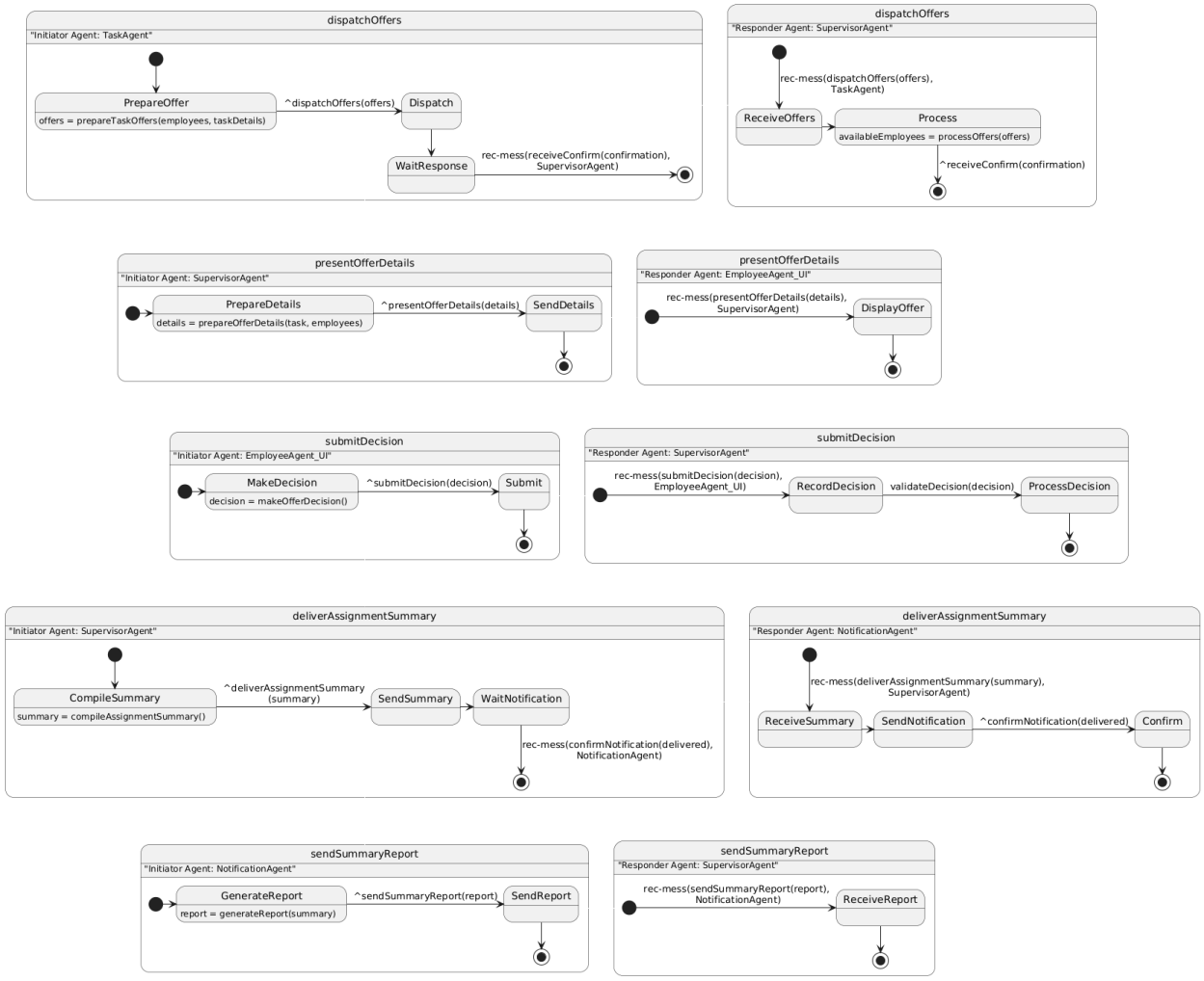

Figure 4.12: LLM-generated Communication Class Diagrams for RMS

Diagram and initialized within the designated containers, as specified in the Deployment Diagram.

While both implementations followed the high-level design models, differences arose in how the agents were implemented at a detailed level. The human-generated code introduced additional supporting classes, primarily for GUI and business logic, that were not defined during the analysis and design phases. Although it did not follow the detailed agent structure outlined in the Agent Architecture Model, the LLM-generated code generally aligned more closely with the design documents.

Overall, the LLM-generated implementation was simpler and contained some syntax errors, but it demonstrated better consistency with the design artefacts. In contrast, the human-generated code provided a more complete implementation but deviated more significantly from its original design.

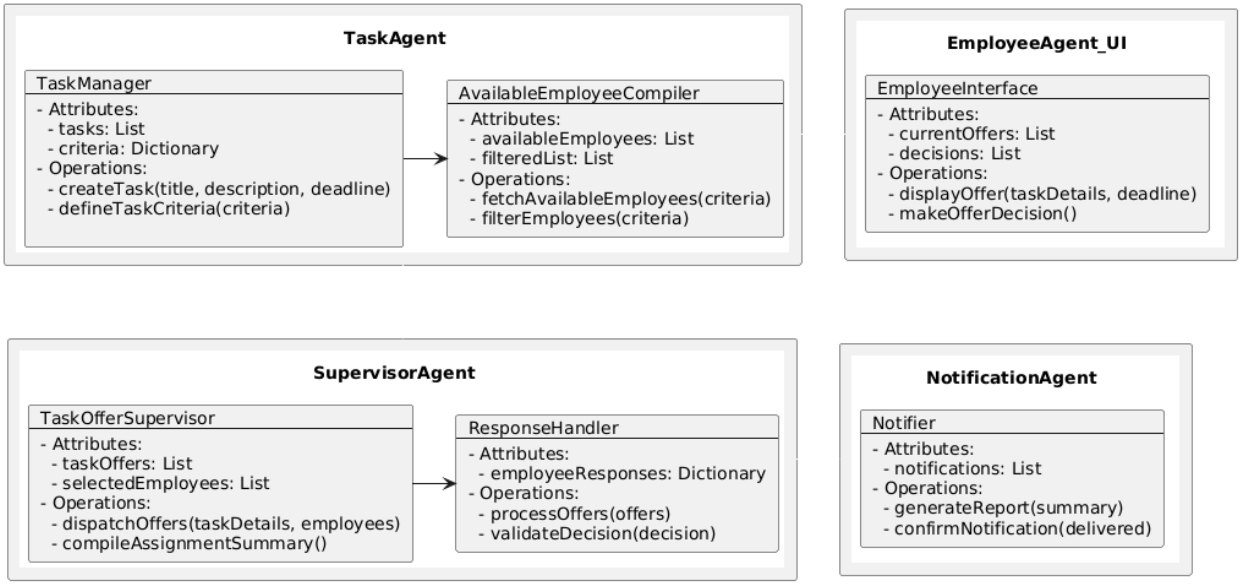

Figure 4.13: LLM-generated Agents' Architecture for RMS

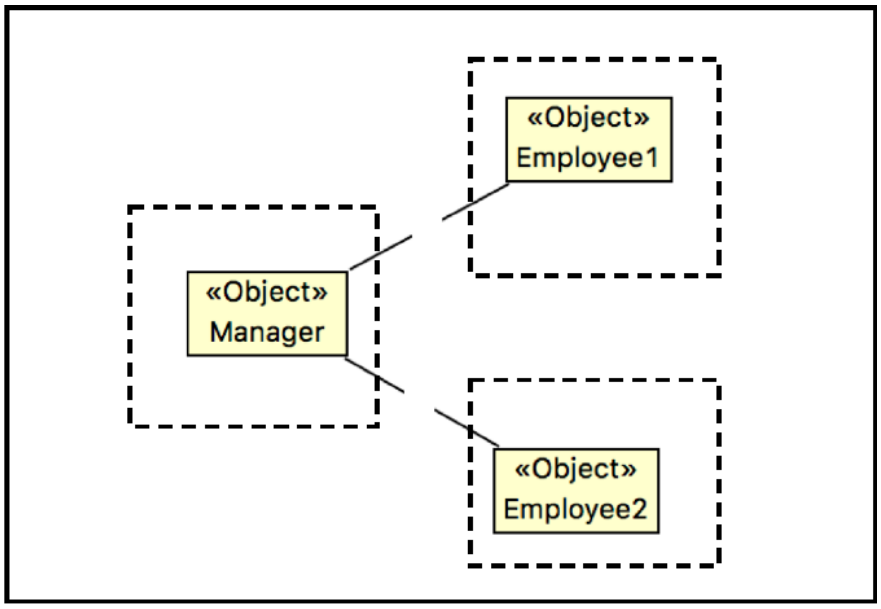

Figure 4.14: Human-generated Deployment Diagram for RMS

4.5 Comparison

After assessing the generated models and code, the following observations were made:

- The LLM-generated results provided a more detailed and partitioned breakdown of the system's functionalities compared to human-generated outputs. For instance, while human-generated use cases primarily focused on the high-level interactions between the actors with a single sequence diagram for

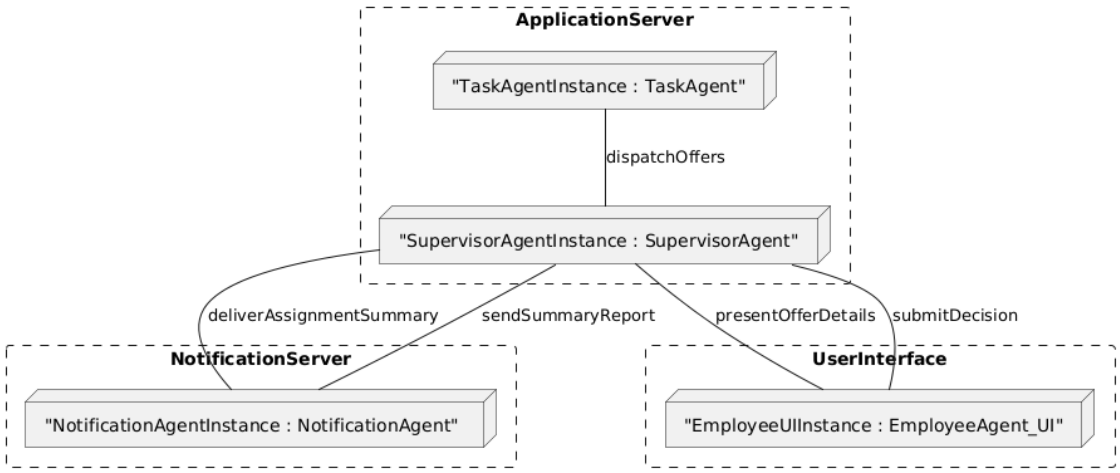

Figure 4.15: LLM-generated Deployment Diagram for RMS

the whole system functionality, the LLM offered more granular and partitioned use cases, including both positive and negative scenarios, and provided more detail at each step. Additionally, the human-generated role model included more general roles like Creator, Collector, Distributor, and Notifier, whereas the LLM specified more specific roles with detailed tasks, such as Task Manager, TaskOffer-Suppervisor, and Employee_UI. As a result, The LLM-generated models were more modular and better aligned with MaSE's modular approach.

- The human-generated models primarily focused on core system functionality, offering less alignment with the requirements. In certain steps, such as the concurrent task model and communication class diagram, the human models addressed only the main system functionality, overlooking the other aspects. In contrast, the LLM covered all the requirements and generated the models as needed.

- When comparing the developed code, the human-generated code included its two agents, EmployeeAgent and ManagerAgent, with additional support from GUI components and business logic. However, aside from these two core agent classes, supporting classes were not defined during the analysis and design phases, leading to a significant gap between design and implementation.

- Despite having some minor syntax errors, the LLM-generated code more closely followed the provided design documents but showed deficiencies in some implementation details regarding the agents' internal architecture.

In general, human-generated results were simpler but less modular and not complete. In contrast, the LLM produced more modular and comprehensive results, performing better in labour-intensive task. While both the human and LLM models seemed to perform similarly in term of model generation, the LLM demon-

strated better performance tasks such as providing enough models for communication class diagram. The LLM-generated models were more detailed, modular, and better aligned with MaSE's modularity requirements.

Regarding the implemented code, while LLM skipped some detailed implementations and contained some syntax errors, it was more closely aligned with analysis and design artefacts. The human-generated code, though more comprehensive in terms of implementation, was significantly different from analysis and design artefacts.

4.6 Extracted Hypotheses

From this case study, we propose the following hypotheses for further experimentation:

1. Does LLM provide the same results as human, and if not, which one performs better.
2. Do different LLMs provide the same results, and which LLM performs better than others.
3. How does LLM perform for projects from different domains. Is there any difference in LLM performance for projects from different domains. Does LLM perform better with more popular domains and projects.
4. How does LLM perform for projects of different sizes. Is there any difference in LLM performance for projects of varying sizes. Does LLM lose track for larger projects.

Chapter 5

Evaluation - Experiment

5.1 Introduction

Comparing human performance with that of Large Language Models (LLMs) in Multi-Agent System (MAS) development forms the basis for a key hypothesis, which this chapter investigates through a structured experimental design.

The results of the experiment compare the performance of three different LLMs against human developer. On one hand, it examines whether LLMs can perform as effectively as human; on the other, it compares the performance of several state-of-the-art LLMs in MAS development.

The experiment comprises nine sub-experiments, each corresponding to a specific step in the LLM-based MaSE framework.

5.2 Experiment preparation

Before diving into the experiment details, it is important to address key considerations common to empirical studies. Our experiment primarily aims to conduct quantitative analysis by quantifying qualitative data. In quantitative analysis, measurement and statistical analysis are the main focus [42].

5.2.1 Measurement

The central part of empirical studies is the measurement [42]. To investigate the effect of certain inputs on the outputs, we must be able to measure both accurately. As described in [42], "A measure is the number or symbol assigned to an entity by this relationship to characterize an attribute". A scale must be defined for measurement, as its type determines the suitable statistical analyses. In practice, we define specific metrics

and assess them through the empirical study. The next two sections describe the metrics, and the scale of measurement used.

Metrics

The outputs of this study are the models and code generated based on the system requirements. Therefore, we define metrics and a scale for measurement to evaluate the quality of these outputs, which reflects the performance of the developer (human or LLM). Inspired by [43], the five following metrics are used to assess the quality:

1. **Completeness (Consistency):** to what extent does the model or code align with the system requirements and previously generated models?
2. **Correctness:** to what extent does the model or code correctly reflect the intended structure and behavior of the system?
3. **Adherence to the standard (syntax, semantic):** To what extent does the model or code adhere to the expected syntax and maintain semantic correctness?
4. **Understandability:** to what extent is the model or code clear and without redundancy?
5. **Terminology alignment:** to what extent does the terminology used in the model or code align with the system requirements and earlier artefacts?

Scale of measurement

As discussed in [44], the Universal Leveled Scale (ULS), a modified version of Steven and Likart scales, was introduced to quantify the qualitative attributes. Although our metrics are discrete in nature, we treat them as representing continuous values with a true zero (indicating absence of the attribute), allowing for more powerful statistical analysis.

ULS is organized in four levels according to their power. We use ULS 3, where values are integers between 0 and 10, representing 0% to 100% of attribute coverage. This scale supports both parametric and non-parametric statistical methods.

5.2.2 Statistical Data Analysis

Statistical data analysis applies mathematical techniques to identify patterns and relationships in collected data. Hypotheses testing forms the foundation of statistical analysis, validating or rejecting assumptions based on evidence.

Statistical techniques fall into two categories: parametric and non-parametric. Parametric methods assume certain conditions, such as normal distribution of data, while non-parametric methods do not. Generally, parametric methods are more powerful when assumptions hold true [4].

Figure 5.1 illustrates the decision process for choosing between parametric and non-parametric approaches.

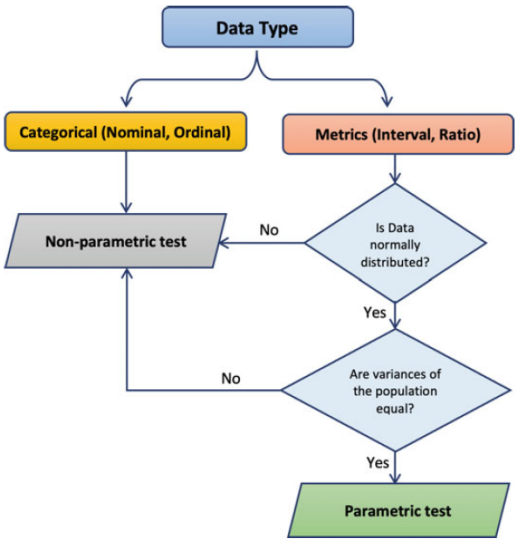

Figure 5.1: Choosing between Parametric and Non-Parametric Methods [4]

Based on our metrics and ULS, we can apply both approaches. When comparing more than two groups (as in our case), and if the data is normally distributed, the Analysis of Variance (ANOVA) test is preferred. For independent observations, one-way ANOVA is used; for dependent observations, a Repeated Measure ANOVA is suitable [4].

If the assumptions of ANOVA are violated (e.g. data is not normally distributed), we apply non-parametric alternatives: Kruskal-Willis for independent data, or Friedman's ANOVA test for dependent data [4].

ANOVA test assumes two alternative hypotheses. The default or null hypothesis (H0), which assumes that there is no significant difference between the groups, and the alternative hypothesis (Ha), which assumes that there is a significant difference between the data groups. If the null hypothesis is rejected, pairwise comparisons are performed using paired t-tests (for parametric data) or Wilcoxon test (for non-parametric data), with multiple-comparison correction, to determine which groups differ significantly. These are considered post-hoc tests, used to identify specific differences between groups [4].

5.3 Experiment process

The experiment followed a structured process consisting of five phases: scoping, planning, operation, analysis and interpretation, and presentation and packaging [42]. Each phase is elaborated below in the context of our study.

5.3.1 Scoping

The first phase involves defining the scope of the experiment. This includes identifying the object (the entity that is being studied like product or process), the purpose (the goal of the experiment), the quality focus (the expected output that has being studied), the perspective (the viewpoint that the result is being interpreted from), and the context (the environment in which the experiment is run) of the study. Putting this together, the scope of our experiment can be defined as:

> *Analyze LLM-based MaSE process*
> *for the purpose of evaluation*
> *with respect to the quality of models and code*
> *from the point of view of a software engineer*
> *in the context of MAS development*

5.3.2 Planning

This phase defines how the experiment is conducted, including the setup of experimental parameters.

Context Selection

This phase involves determining four dimensions for the context (offline vs. online or real life, student vs. professional, toy vs. real problems, specific vs. general). The experiment is going to be offline, using projects from Agent-oriented software engineering course that were done by graduate students at the University of Calgary. These projects represent toy problems and are specific to MAS development.

Hypothesis Formulation

We define both null and alternative hypotheses:

- ***Null Hypothesis (H0):*** *There is no significant difference in performance between human and LLMs in MAS development using MaSE.*
- ***Alternative Hypothesis (Ha):*** *There is a significant difference in performance between human and LLMs in MAS development using MaSE.*

We aim to compare the performance of different state-of-the-art LLMs, utilised the LLM-based MaSE, with human, utilised manual approach, for MAS development. If differences exist, either among LLMs or between LLMs and human developers, they will be captured statistically.

Variable Selection

Two types of variables are involved in an experiment:

- **Independent variables:** Inputs that can be controlled or changed during the experiment and are expected to influence the outcome. The changing variable is called factor and different values for the factor are called treatments.
- **Dependent variables:** Outputs that are measured to evaluate the effect of the independent variables. It is generally recommended to select only one dependent variable to reduce the risk of fishing and error rate threat to conclusion validity.

In our experiment, the input (independent variable) is the MAS development project, which consists of developing a simple shopping system. This project was intentionally chosen from a very well-known domain to ensure a common background and contextual understanding for both human and LLMs.

The treatments applied to this variable consist of one human-based development process and three LLM-based approaches. The LLMs were selected from state-of-the-art LLM families that offer free access via a chatbot or API. The following models, each representing the latest version of their respective families at the time of the experiment, were chosen:

- GPT 4.1
- Gemini 2.5 Pro Preview
- DeepSeek R1

The output (dependent variable) is the performance of each development process. The performance is evaluated using the five predefined qualified metrics defined in Section 5.2.1.

Selection of the subjects (sample from the population)

This involves determining the sample size, which impacts the generalizability of results. In this experiment, the evaluation of the MAS development outputs was performed by the author (a graduate student with knowledge of MAS concepts), along with other LLMs acting as evaluators.

The use of LLMs as evaluators, commonly referred to as the LLM-as-a-judge approach, has gained popularity in recent years, as these models can effectively approximate human preferences in open-ended tasks [45].

Since the outputs (models and code) were in textual format and followed specific templates provided in the prompts, the human-generated outputs were manually converted into the same format. This enabled the LLMs to conduct the evaluation process for human-generated outputs effectively.

Design

An experiment is a set of tests, and each test is a combination of an object, subject, and treatment. In this stage, the experiment is designed by determining the number of tests and replications, and by defining the statistical analysis methods that can potentially reject the hypothesis.

In our case, we have one object (the MAS project), four treatments (human and three LLMs), and four subjects (four evaluators or raters). All treatments are applied to the same object, and all subjects evaluate the resulting outputs.

Ideally, to achieve independent observations required for one-way ANOVA, 16 raters (4 treatments $\times$ 4 replications) would be needed. However, since only four raters are available, the observations are dependent. Therefore, Repeated Measures ANOVA (RM ANOVA) and Friedman's ANOVA are used as parametric and non-parametric methods, respectively, for hypothesis testing. For post-hoc pairwise comparisons, we apply the paired t-tests for parametric analysis and the Wilcoxon test for non-parametric analysis [4].

Instruments

The instruments used in an experiment fall into three categories: objects, guidelines, and measurement. The object of the experiment (MAS project) is specially drawn from a course project in the Agent-Oriented Software Engineering class at the University of Calgary. This course material, including the students' final report and implementations, provides the human-generated outputs necessary for comparison.

Evaluation measurements and guidelines were developed based on ULS 3 described earlier, ensuring a standardized and quantifiable approach to assessing quality. These guidelines were followed by the human rater and also embedded into prompts given to the LLM judges, ensuring consistency in the evaluation process. The scoring mechanism, aligned with the predefined metrics and measurement scale, served as the foundation for both human and LLM assessments. The prompt templates used for LLM-based evaluation are included in Attachment A.

Validity Evaluation

To ensure the validity of the collected data, a consistent evaluation framework was applied across all treatments. All outputs, whether produced by humans or LLMs, were formatted uniformly according to predefined templates, allowing for unbiased comparison. Evaluation criteria were clearly defined and applied by both the human evaluator and the LLM judges using standardized prompts. This uniformity in input format, scoring rubric, and evaluation procedure helps ensure that the resulting data accurately reflects differences in performance and is suitable for reliable statistical analysis.

5.3.3 Operation

The operation phase involves three sub-steps: preparation, execution and data validation. The experiment was executed according to the defined plan, and the experiment results were collected.

As the LLM-based MaSE framework continues through its distinct steps, for each step, a sub-experiment is done. For every sub-experiment, the following evaluation procedure was followed:

1. Begin with the first step of the framework.
2. Generate and collect the outputs produced by each treatment for that step.
3. Evaluate the outputs using both a human rater and LLM judges.
4. Conduct statistical analysis to determine whether the null hypothesis should be rejected.
5. If the null hypothesis is not rejected, conclude that there is no significant difference between the treatments. Go to step 8.
6. If the null hypothesis is rejected, it indicates a significant difference between treatments.
7. Perform pairwise comparisons to identify which treatments performed better or worse.
8. If all framework steps are not completed, go to step 2.

9. The process is finished.

This procedure is summarized in Figure 5.2, which illustrates the full evaluation loop across the framework steps.

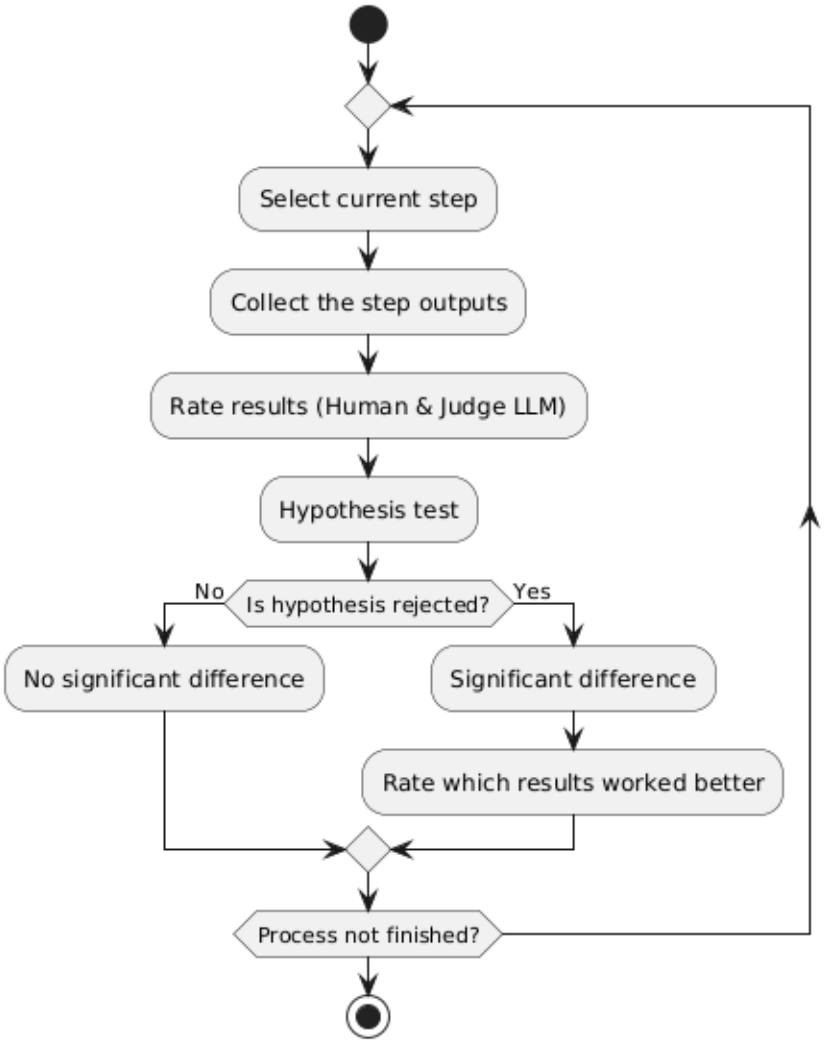

Figure 5.2: Evaluation Process

To maintain consistency, all LLMs (with clear history) were prompted using the same project data, directly extracted from the original student project. Outputs were collected and evaluated using the predefined metrics and measurement guidelines described earlier. These results are demonstrated and analyzed in the next section.

5.3.4 Analysis and interpretation

In this phase, the collected data is statistically analyzed and interpreted to test the experiment hypotheses. For each step in the MaSE methodology, the raw data from evaluation process are presented, using the five evaluation metrics defined earlier (referred as M1 to M5).

The four treatments (human, GPT 4.1, Gemini 2.5 pro preview, and DeepSeek R1) are labeled as T1 to T4, respectively. The four evaluators (human and other three LLMs) are referenced as S1 to S4.

The statistical analysis process begins with summarizing the raw evaluation data (see Figure 5.3). All statistical computations, including assumption checks and hypothesis testing, were performed using Python statistical libraries.

Before applying Repeated Measure ANOVA (RM ANOVA), we first check whether its assumptions are met:

1. **Normality and Outliers:** We need to ensure that the data is approximately normally distributed. This can be tested through a "Normal probability plot (Q-Q plot) of residuals". If the points align with the expected diagonal, the normality assumption is satisfied, and no extreme outliers are present.

2. **Homogeneity of Error Variances:** The variances of data groups should be equal. This can be tested by "Residuals against fitted values plot". If the residuals are randomly distributed without a visible pattern, the variances are considered equal.

3. **Sphericity:** this assumption is specific to RM ANOVA and refers to equal variances in the differences between all treatment pairs within each subject. This condition usually is tested using Mauchly's test. If Mauchly's test result (p-value) is greater than 0.05, the assumption holds.

If all three assumptions are satisfied, RM ANOVA is applied to determine whether there are significant differences between treatments. If any assumption is violated, we use Friedman's ANOVA, a non-parametric alternative suitable for dependent observations.

In cases where the null hypothesis is rejected, pairwise comparisons are conducted to identify which specific treatments differ. For parametric data, paired t-tests are applied; for non-parametric data, Wilcoxon tests are used. The results from these post-hoc tests help determine which treatment (if any) significantly outperformed others. Treatment performance can then be ranked based on the average scores and statistical outcomes.

The following subsections present step-by-step analysis for each MaSE framework component, beginning with Goal Hierarchy.

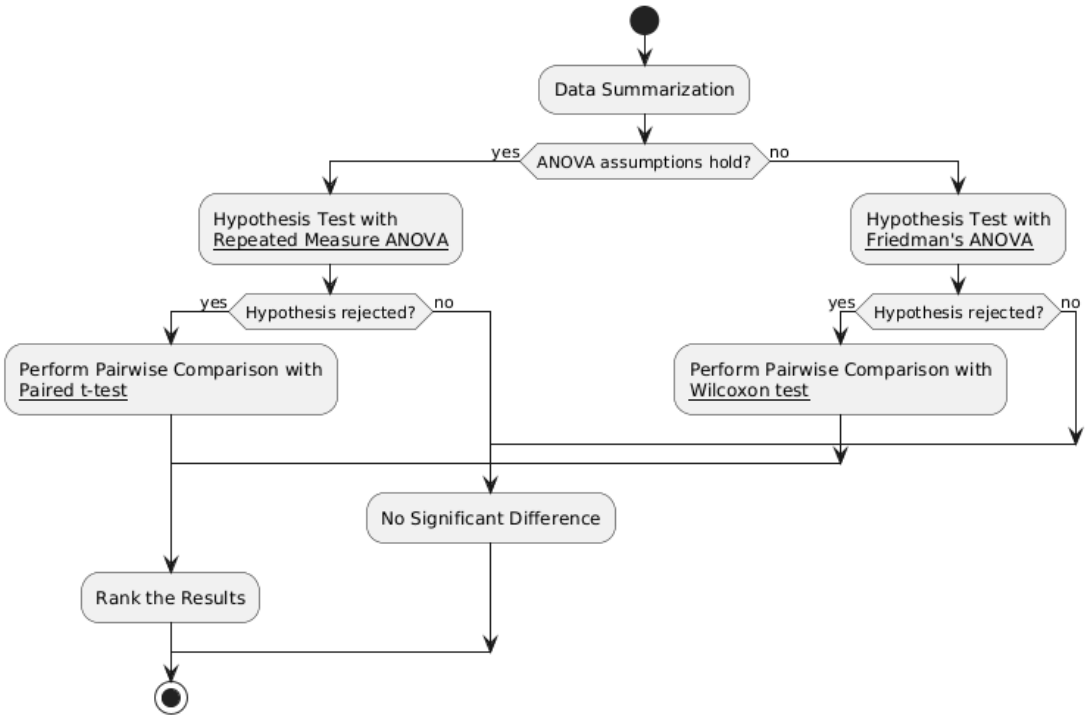

Figure 5.3: Statistical Analysis Process

Analysis: Goal Hierarchy

Raw Data

Table 5.1 presents the raw evaluation scores for the Goal Hierarchy step, assessed across five metrics (M1-M5) by four subjects (S1-S4) for each treatment (T1-T4).

Data summarization

To prepare for statistical analysis, the average of individual metric scores for each subject-treatment pair were calculated. The summarized data is shown in Table 5.2.

ANOVA Assumptions Check

Before applying Repeated Measure ANOVA, assumptions were evaluated using Python statistical libraries. Normality was examined through a Q-Q plot of residuals (Figure 5.4), which showed that the data closely followed the expected distribution, indicating no major deviations or outliers. Homogeneity of variances was evaluated by plotting residuals against fitted values (Figure 5.5); the absence of a clear pattern in the residuals confirmed that group variances were approximately equal. Finally, sphericity was tested using Mauchly's test, which produced a p-value of 0.1625. Since this value is greater than 0.05, the assumption of sphericity was considered satisfied. With all three conditions met, it was appropriate to proceed with RM ANOVA.

Subject	Metric	T1	T2	T3	T4
S1	M1	8	10	10	10
	M2	10	9	10	10
	M3	10	10	10	10
	M4	10	10	10	10
	M5	10	10	10	10
S2	M1	8	10	10	10
	M2	7	10	10	10
	M3	8	9	10	10
	M4	9	10	10	10
	M5	8	10	10	10
S3	M1	7	10	9	9
	M2	7	9	10	9
	M3	9	10	10	10
	M4	8	9	10	10
	M5	9	10	10	10
S4	M1	8	10	10	9
	M2	9	10	10	10
	M3	9	9	10	10
	M4	10	10	9	9
	M5	10	10	10	10

Table 5.1: Raw Data for Goal Hierarchy - Step 1.

Subject	T1	T2	T3	T4
S1	9.6	9.8	10	10
S2	8	9.8	10	10
S3	8	9.6	9.8	9.6
S4	9.2	9.8	9.8	9.6
Mean	8.7	9.75	9.9	9.8

Table 5.2: Summarized Data for Goal Hierarchy - Step 1.

RM ANOVA Test

The analysis of variance with repeated measurement calculates a p-value for the data and if p-value is smaller than the predefined significance level (normally 0.05), the null hypothesis is rejected.

Source of Variation	SS (Sum of Squares)	df (Degrees of Freedom)	MS (Mean Square)	F-value	P-value
Treatment (Between Groups)	4.860	3	1.620	9.7525	0.0035
Error (Within Groups)	1.495	9	0.166		
Total	6.355	12			

Table 5.3: RM ANOVA Calculation Results for Goal Hierarchy - Step 1.

Table 5.3 summarizes the RM ANOVA output. Since the p-value is below 0.05, the null hypothesis is rejected, indicating a significant difference between the treatments.

Pairwise Comparison

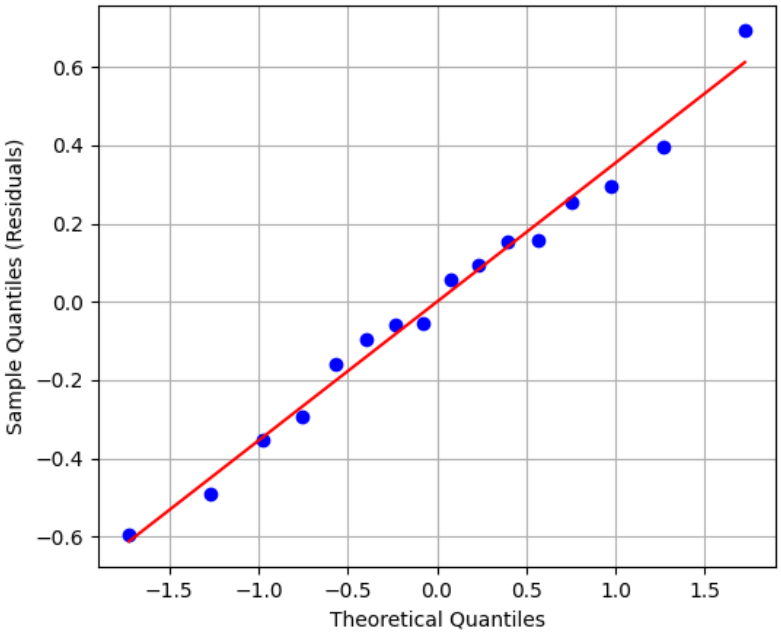

Figure 5.4: Normal Probability Plot of Residuals for Goal Hierarchy - Step 1

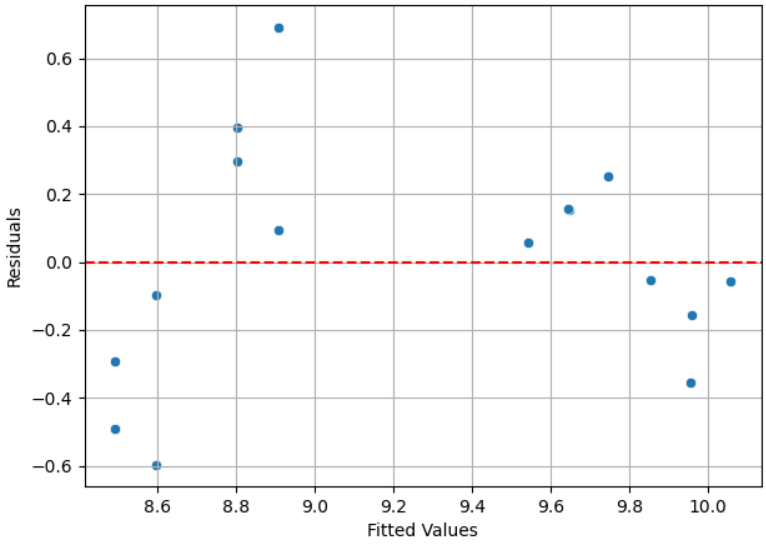

Figure 5.5: Residual Plot against Fitted Values for Goal Hierarchy - Step 1

To further investigate which treatments differ, pairwise comparisons were conducted. The results are shown in Table 5.4.

Although RM ANOVA showed a significant difference overall, the post-hoc pairwise test did not reveal

Treatment1	Treatment2	T	p-unc	p-corr	BF10	Hedges' g	$p - unc < 0.05$	$p - corr < 0.05$
T1	T2	0	1	1	0.428	0	No	No
T1	T3	-5.9797	0.0094	0.0562	7.347	-2.9623	Yes	No
T1	T4	-4.5399	0.02	0.06	4.421	-3.0382	Yes	No
T2	T3	-2.7187	0.0726	0.1161	1.848	-1.5545	No	No
T2	T4	-2.6441	0.0774	0.1161	1.770	-1.6705	No	No
T3	T4	-1	0.391	0.4692	0.613	-0.5693	No	No

Table 5.4: Paired T-Test Results for Goal Hierarchy - Step 1.

statistically significant pairwise differences after correction. This may be due to the correction method, which adjusts for multiple comparisons and increases the threshold for detecting significance. As a result, some differences, particularly between T1 and T3, and between T1 and T4, were not statistically confirmed despite having low uncorrected p-values. These results suggest potentially meaningful differences that could reach significance with a larger sample size. Additionally, large Hedges' g and Bayes Factors for these pairs indicate practically important differences.

The treatments can be ranked by average score as: $T3 > T4 > T2 > T1$. Although not statistically significant after correction, results suggest that Gemini (T3) and DeepSeek (T4) may outperform the human (T1) in generating the Goal Hierarchy model. With a larger sample size, these trends could likely achieve statistical significance.

Analysis: Sequence Diagram

The raw evaluation scores for the Sequence Diagram step are shown in Table 5.5, and the summarized scores per subject are presented in Table 5.6.

ANOVA Assumptions Check

Assumption checks for RM ANOVA were conducted using Python. The Q-Q plot (Figure 5.6) indicated that the residuals followed a normal distribution, while the residuals vs. fitted values plot (Figure 5.7) confirmed the homogeneity of variances. Mauchly's test resulted a p-value of 0.421, satisfying the sphericity condition.

RM ANOVA Test and Pairwise Comparison

The RM ANOVA results (Table 5.7) showed a p-value below 0.05, indicating a statistically significant difference in performance between treatments. However, as shown in the paired t-test results (Table 5.8), none of the comparisons reached statistical significance, even before correction.

Although the mean scores suggest that Gemini (T3) slightly outperformed the others, the lack of significant pairwise differences means there is not enough evidence to conclude that any specific treatment performed better in a statistically meaningful way for this step. The treatments can be roughly ranked as

Subject	Metric	T1	T2	T3	T4
S1	M1	10	10	10	10
	M2	10	9	10	9
	M3	10	10	10	10
	M4	9	10	10	10
	M5	10	10	10	10
S2	M1	10	10	10	10
	M2	10	10	10	10
	M3	9	10	10	10
	M4	9	10	10	10
	M5	9	10	10	10
S3	M1	8	8	9	9
	M2	7	8	10	9
	M3	8	9	10	10
	M4	8	9	10	10
	M5	7	10	10	10
S4	M1	9	9	10	8
	M2	8	10	10	10
	M3	7	10	10	9
	M4	8	9	9	10
	M5	9	10	10	10

Table 5.5: Raw Data for Sequence Diagram – Step 2.

Subject	T1	T2	T3	T4
S1	9.8	9.8	10	9.8
S2	9.4	10	10	10
S3	7.6	8.8	9.8	9.6
S4	8.2	9.6	9.8	9.4
Mean	8.75	9.55	9.9	9.7

Table 5.6: Summarized Data for Sequence Diagram – Step 2.

Source of Variation	SS	df	MS	F-value	P-value
Treatment (Between Groups)	3.05	3	1.017	5.112	0.0246
Error (Within Groups)	1.79	9	0.199		
Total	4.84	12			

Table 5.7: RM ANOVA Calculation Results for Sequence Diagram – Step 2.

Treatment1	Treatment2	T	p-unc	p-corr	BF10	Hedges' g	$p - unc < 0.05$	$p - corr < 0.05$
T1	T2	-2.5298	0.0854	0.169	1.655	-0.8541	No	No
T1	T3	-2.5145	0.0866	0.169	1.640	-1.3715	No	No
T1	T4	-2.2238	0.1126	0.169	1.371	-1.1056	No	No
T2	T3	-1.5785	0.2126	0.2551	0.897	-0.7993	No	No
T2	T4	-0.6765	0.5472	0.5472	0.512	-0.3148	No	No
T3	T4	2.4495	0.0917	0.169	1.577	0.8696	No	No

Table 5.8: Paired T-Test Results for Sequence Diagram – Step 2.

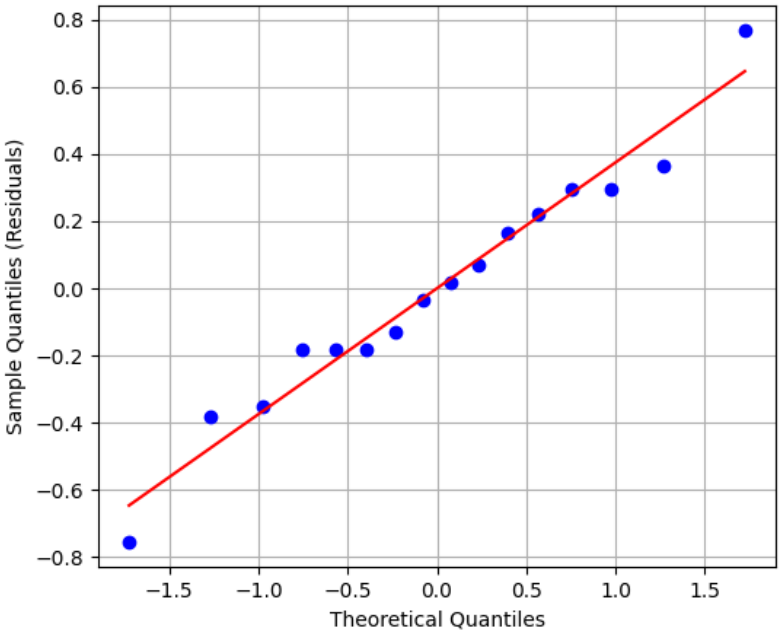

Figure 5.6: Normal Probability Plot of Residuals for Sequence Diagram - Step 2

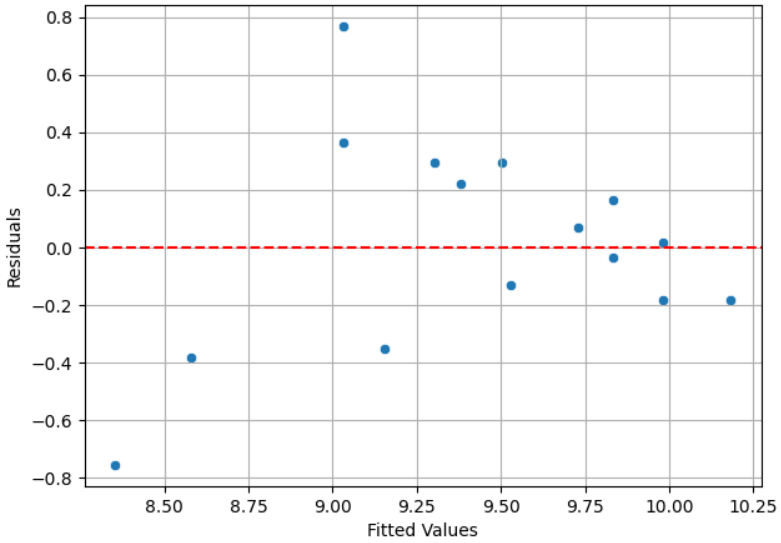

Figure 5.7: Residual Plot against Fitted Values for Sequence Diagram - Step 2

$T3 > T4 > T2 > T1$ based on average scores.

Analysis: Role Model

Table 5.9 contains the raw evaluation data for the Role Model step, and Table 5.10 provides the summarized scores.

Subject	Metric	T1	T2	T3	T4
S1	M1	8	8	10	10
	M2	10	8	10	10
	M3	9	10	10	10
	M4	8	9	10	10
	M5	10	10	10	10
S2	M1	8	10	10	10
	M2	7	10	10	9
	M3	7	10	10	10
	M4	8	10	10	10
	M5	7	10	10	9
S3	M1	5	7	10	6
	M2	4	6	10	8
	M3	6	8	10	7
	M4	5	6	9	7
	M5	6	8	10	9
S4	M1	7	8	10	7
	M2	6	9	10	8
	M3	6	8	10	10
	M4	7	7	8	8
	M5	8	10	10	9

Table 5.9: Raw Data for Role Model – Step 3.

Subject	T1	T2	T3	T4
S1	9	9	10	10
S2	7.4	10	10	9.6
S3	5.2	7	9.8	7.4
S4	6.8	8.4	9.6	8.4
Mean	7.1	8.6	9.85	8.85

Table 5.10: Summarized Data for Role Model – Step 3.

ANOVA Assumptions Check

Assumption checks passed. the Q-Q plot (Figure 5.8) and residuals plot (Figure 5.9) did not indicate any violations, and Mauchly's test returned a p-value of 0.4261.

RM ANOVA Test and Pairwise Comparison

The RM ANOVA test (Table 5.11) confirmed a significant difference between treatments ($p < 0.05$).

Pairwise comparison results (Table 5.12) revealed no statistically significant differences after correction. However, the comparisons between T1 and T3, and T1 and T4, showed low uncorrected p-values and large Hedges' g, suggests practically meaningful differences.

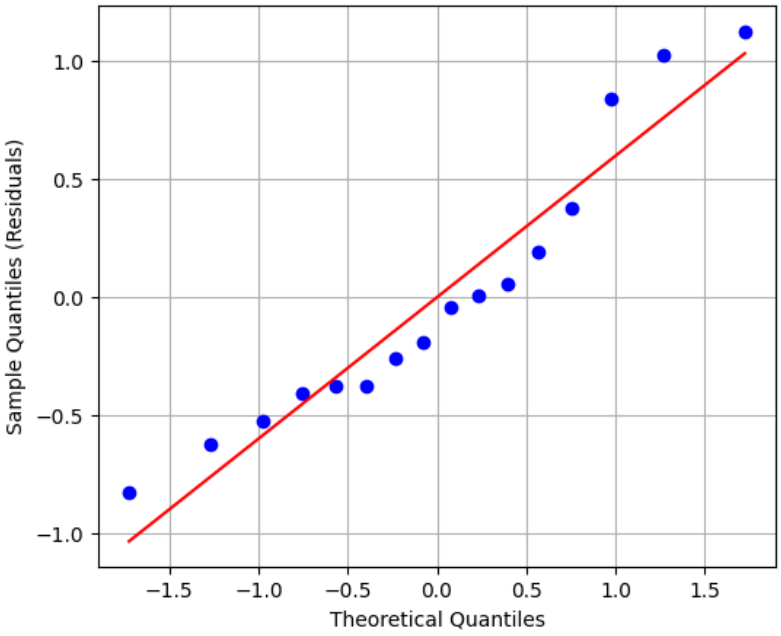

Figure 5.8: Normal Probability Plot of Residuals for Role Model - Step 3

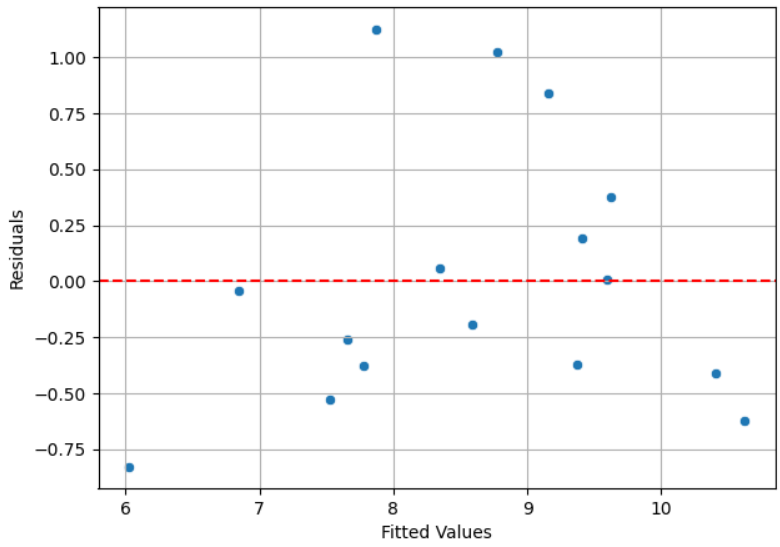

Figure 5.9: Residual Plot against Fitted Values for Role Model - Step 3

Based on average scores, treatments may be ranked as $T3 > T4 > T2 > T1$, with LLMs, particularly Gemini and DeepSeek, potentially outperforming the human baseline in Role Model creation.

Source of Variation	SS	df	MS	F-value	P-value
Treatment (Between Groups)	15.50	3	5.167	9.529	0.00373
Error (Within Groups)	4.88	9	0.542		
Total	20.38	12			

Table 5.11: RM ANOVA Calculation Results for Role Model – Step 3.

Treatment1	Treatment2	T	p-unc	p-corr	BF10	Hedges' g	$p - unc < 0.05$	$p - corr < 0.05$
T1	T2	-2.754	0.0705	0.141	1.886	-0.9177	No	No
T1	T3	-3.7336	0.0335	0.1005	3.125	-2.1374	Yes	No
T1	T4	-6.0927	0.0089	0.0533	7.610	-1.0949	Yes	No
T2	T3	-2.157	0.1199	0.1799	1.315	-1.2115	No	No
T2	T4	-0.8372	0.4639	0.4639	0.557	-0.1784	No	No
T3	T4	1.8898	0.1552	0.1862	1.105	1.0272	No	No

Table 5.12: Paired T-Test Results for Role Model – Step 3.

Analysis: Concurrent Task Model

Raw data and summary statistics are presented in Table 5.13 and Table 5.14, respectively.

Subject	Metric	T1	T2	T3	T4
S1	M1	6	8	10	10
	M2	7	7	10	8
	M3	9	10	10	10
	M4	6	9	10	10
	M5	10	10	10	10
S2	M1	7	10	10	10
	M2	8	10	10	10
	M3	7	10	10	10
	M4	8	10	10	10
	M5	8	10	10	10
S3	M1	6	7	10	9
	M2	4	5	10	9
	M3	6	8	10	10
	M4	4	6	9	9
	M5	6	7	10	10
S4	M1	4	9	10	6
	M2	5	10	10	7
	M3	6	10	10	10
	M4	5	8	7	9
	M5	7	10	10	10

Table 5.13: Raw Data for Concurrent Task Model – Step 4.

ANOVA Assumptions Check

Assumption checks were passed (Figure 5.10 and Figure 5.11; Mauchly's p = 0.5831).

RM ANOVA Test and Pairwise Comparison

The RM ANOVA test results (Table 5.15) showed a highly significant difference among treatments (p =

Subject	T1	T2	T3	T4
S1	7.6	8.8	10	9.6
S2	7.6	10	10	10
S3	5.2	6.6	9.8	9.4
S4	5.4	9.4	9.4	8.4
Mean	6.45	8.7	9.8	9.35

Table 5.14: Summarized Data for Concurrent Task Model – Step 4.

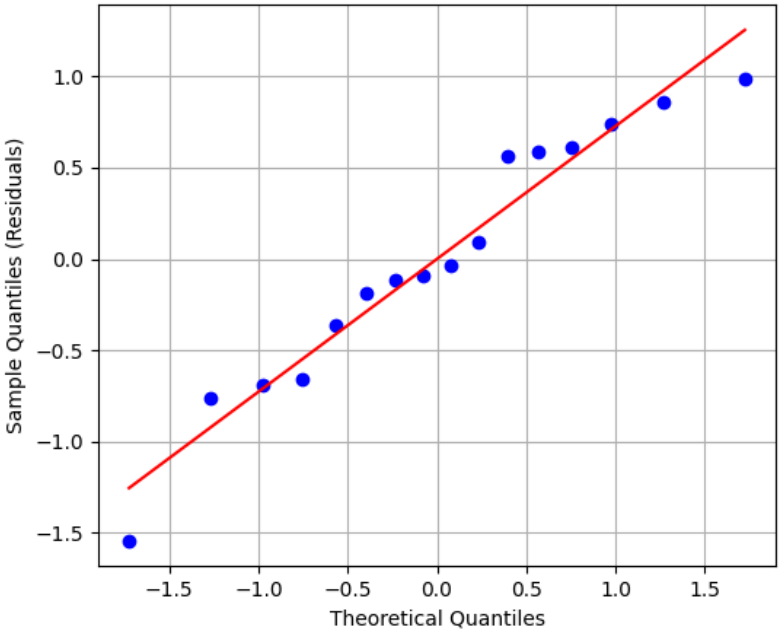

Figure 5.10: Normal Probability Plot of Residuals for Concurrent Task Model - Step 4

0.0017).

Source of Variation	SS	df	MS	F-value	P-value
Treatment (Between Groups)	26.53	3	8.843	11.9684	0.0017
Error (Within Groups)	6.65	9	0.739		
Total	33.18	12			

Table 5.15: RM ANOVA Calculation Results for Concurrent Task Model – Step 4.

The results of the paired t-tests (Table 5.16) confirmed statistically significant differences between human treatment (T1) and both LLM-based treatments, Gemini (T3) and DeepSeek (T4), even after correction.

This step provides strong statistical evidence that LLMs outperformed the human baseline. The treatments can be confidently ranked as $T3 > T4 > T2 > T1$.

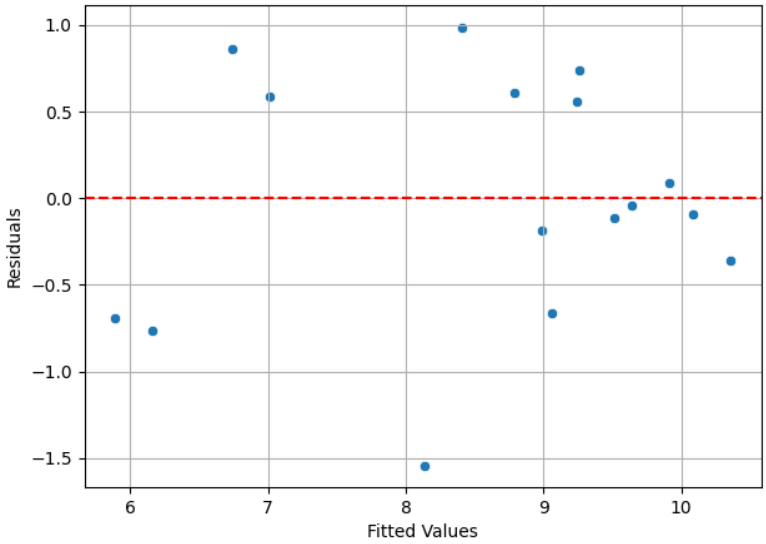

Figure 5.11: Residual Plot against Fitted Values for Concurrent Task Model - Step 4

Treatment1	Treatment2	T	p-unc	p-corr	BF10	Hedges' g	$p-unc<0.05$	$p-corr<0.05$
T1	T2	-3.5175	0.039	0.078	2.82	-1.3887	Yes	No
T1	T3	-5.961	0.0094	0.0283	7.304	-3.0288	Yes	Yes
T1	T4	-6.0469	0.0091	0.0283	7.503	-2.3864	Yes	Yes
T2	T3	-1.457	0.2412	0.2894	0.826	-0.8959	No	No
T2	T4	-0.8067	0.4788	0.4788	0.548	-0.4898	No	No
T3	T4	2.1828	0.117	0.1756	1.336	0.7508	No	No

Table 5.16: Paired T-Test Results for Concurrent Task Model – Step 4.

Design: Agent Class Diagram

Evaluation scores and summaries are shown in Table 5.17 and Table 5.18.

ANOVA Assumptions Check

The assumption checks revealed violations of normality and homogeneity of variances, as seen in Figure 5.12 and Figure 5.13. The normality is violated by the deviation in the lower tails of the Q-Q plot, and the homoscedasticity is violated by the very large negative residual for one fitted value, suggesting unequal variance. Mauchly's test (p = 0.0904) also indicated a violation of sphericity.

Friedman's ANOVA Test and Wilcoxon Pairwise Comparison

The Friedman's ANOVA test was used as the RM ANOVA assumptions were not met. The test results (Table 5.19) indicated a significant difference between treatments ($p-value<0.05$). However, the Wilcoxon pairwise comparisons (Table 5.20) did not reveal any statistically significant differences between specific treatment pairs.

Subject	Metric	T1	T2	T3	T4
S1	M1	8	10	10	10
	M2	10	8	7	10
	M3	7	10	10	10
	M4	10	9	10	10
	M5	10	10	10	10
S2	M1	9	10	10	10
	M2	9	10	10	10
	M3	8	10	10	10
	M4	9	10	10	10
	M5	10	10	10	10
S3	M1	6	6	10	10
	M2	7	5	10	10
	M3	7	8	10	10
	M4	7	5	10	10
	M5	8	7	10	10
S4	M1	8	9	10	9
	M2	7	10	10	10
	M3	8	9	10	10
	M4	9	10	9	9
	M5	9	10	10	10

Table 5.17: Raw Data for Agent Class Diagram – Step 5.

Subject	T1	T2	T3	T4
S1	9	9.4	9.4	10
S2	9	10	10	10
S3	7	6.2	10	10
S4	8.2	9.6	9.8	9.6
Mean	8.3	8.8	9.8	9.9

Table 5.18: Summarized Data for Agent Class Diagram – Step 5.

Source of Variation	Chi-squared statistic	P-value
Treatment (Between Groups)	9.8571	0.0198

Table 5.19: Friedman's ANOVA Calculation Results for Agent Class Diagram – Step 5.

Treatment1	Treatment2	p-unc	p-adj	$p - adj < 0.05$
T1	T2	0.375	0.921	No
T1	T3	0.125	0.750	No
T1	T4	0.125	0.750	No
T2	T3	0.230	0.921	No
T2	T4	0.230	0.921	No
T3	T4	1.000	1.000	No

Table 5.20: Pairwise Wilcoxon Results for Agent Class Diagram – Step 5.

While Friedman's ANOVA indicated a significant difference among the treatments, based on Wilcoxon tests, there is not enough evidence to identify any specific pairs of treatments as being statistically significantly different. Maybe this happened because, as the tool gave a warning, the sample size is too small.

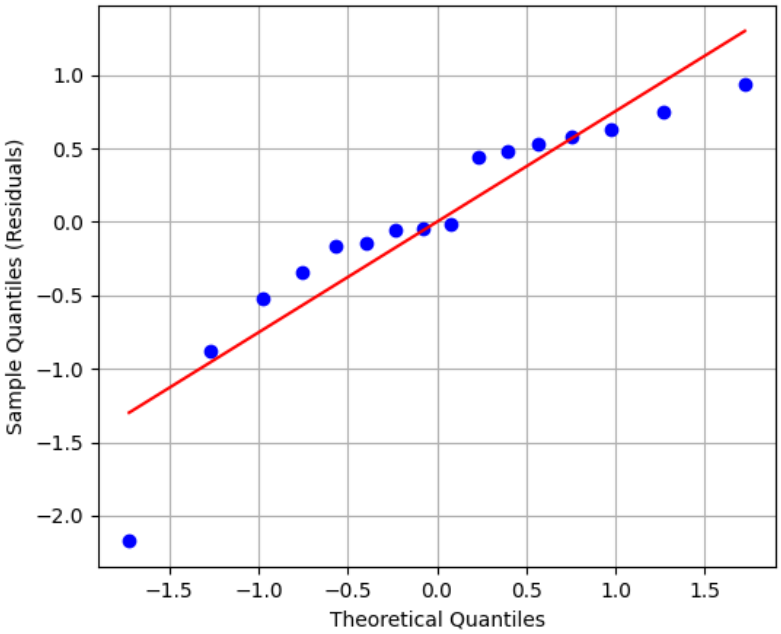

Figure 5.12: Normal Probability Plot of Residuals for Agent Class Diagram - Step 5

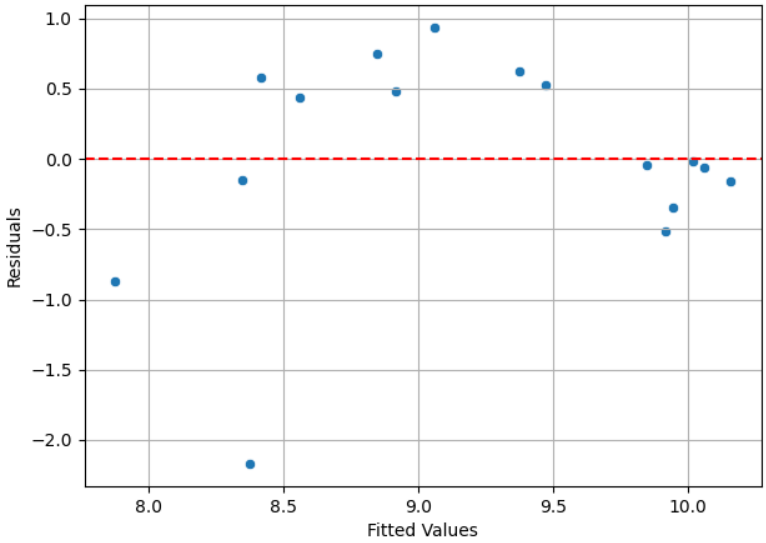

Figure 5.13: Residual Plot against Fitted Values for Agent Class Diagram - Step 5

Based on average scores, the treatments can be ordered as $T4 > T3 > T2 > T1$, but no clear superiority can be established statistically.

Design: Communication Class Diagrams

The raw and summarized data are shown in Table 5.21 and Table 5.22.

Subject	Metric	T1	T2	T3	T4
S1	M1	6	8	10	10
	M2	10	8	7	7
	M3	10	10	10	10
	M4	10	8	10	10
	M5	10	10	10	10
S2	M1	7	10	10	10
	M2	10	10	10	10
	M3	9	10	10	10
	M4	9	10	10	10
	M5	9	10	10	10
S3	M1	9	7	10	10
	M2	9	6	10	9
	M3	9	9	9	10
	M4	10	6	8	9
	M5	10	8	10	10
S4	M1	2	10	10	7
	M2	5	10	8	8
	M3	6	10	9	9
	M4	6	9	10	8
	M5	8	10	10	10

Table 5.21: Raw Data for Communication Class Diagrams – Step 6.

Subject	T1	T2	T3	T4
S1	9.2	8.8	9.4	9.4
S2	8.8	10	10	10
S3	9.4	7.2	9.4	9.6
S4	5.4	9.8	9.4	8.4
Mean	8.2	8.95	9.55	9.35

Table 5.22: Summarized Data for Communication Class Diagrams – Step 6.

ANOVA Assumptions Check

Assumption checks revealed violations of normality and variance homogeneity (Figure 5.14 and Figure 5.15), due to several influential data points that act as outliers. Mauchly's test, though is not significant ($p = 0.1060$).

Friedman's ANOVA Test

Friedman's ANOVA results (Table 5.23) showed no significant differences between treatments. This indicates that all four approaches performed similarly on this step, and the null hypothesis cannot be rejected.

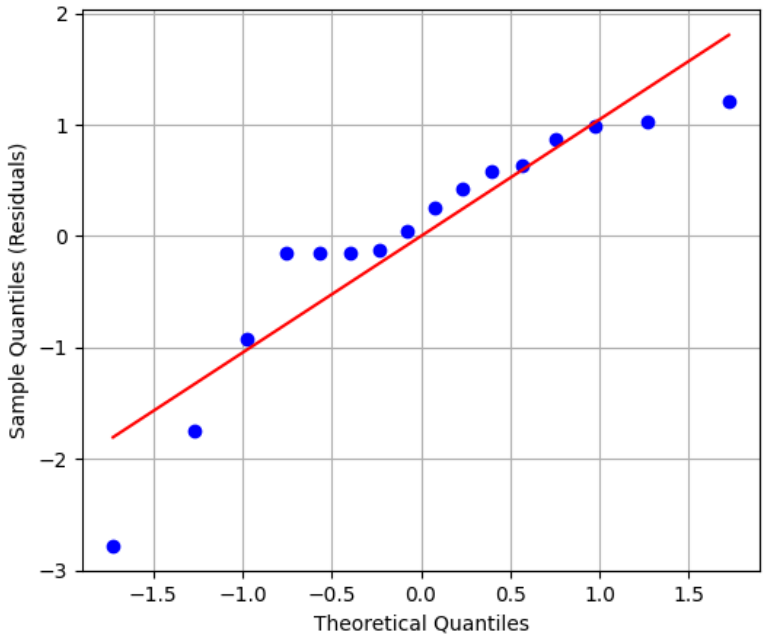

Figure 5.14: Normal Probability Plot of Residuals for Communication Class Diagrams - Step 6

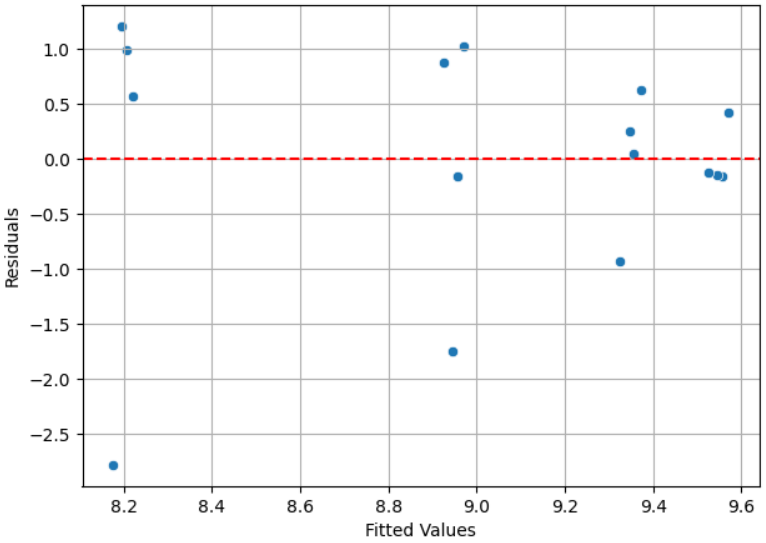

Figure 5.15: Residual Plot against Fitted Values for Communication Class Diagrams - Step 6

Design: Agent Architecture

The data is provided in Table 5.24 and Table 5.25.

ANOVA Assumptions Check

Source of Variation	Chi-squared statistic	P-value
Treatment (Between Groups)	4.1471	0.2460

Table 5.23: Friedman's ANOVA Calculation Results for Communication Class Diagrams – Step 6.

Subject	Metric	T1	T2	T3	T4
S1	M1	8	10	10	10
	M2	9	8	10	10
	M3	9	10	10	10
	M4	7	10	9	9
	M5	10	10	10	10
S2	M1	6	10	10	10
	M2	8	10	10	10
	M3	7	10	10	10
	M4	7	10	10	10
	M5	9	10	10	10
S3	M1	3	6	10	10
	M2	4	5	10	10
	M3	6	8	10	10
	M4	4	5	9	10
	M5	6	7	10	10
S4	M1	3	9	10	10
	M2	4	10	10	9
	M3	5	9	10	10
	M4	6	10	9	10
	M5	7	10	10	10

Table 5.24: Raw Data for Agent Architecture – Step 7.

Subject	T1	T2	T3	T4
S1	8.6	9.6	9.8	9.8
S2	7.4	10	10	10
S3	4.6	6.2	9.8	10
S4	5	9.6	9.8	9.8
Mean	6.4	8.85	9.85	9.9

Table 5.25: Summarized Data for Agent Architecture – Step 7.

Assumption checks showed deviations from normality and variance homogeneity, with Figure 5.16 and Figure 5.17 confirming violations. The assumptions appear to be violated due to two influential data points that act as outliers. Mauchly's test ($p = 1.000$) showed no violation of sphericity, but due to the other issues, Friedman's ANOVA was applied.

Friedman's ANOVA Test and Wilcoxon Pairwise Comparison

The Friedman's ANOVA test results (Table 5.26) indicated a statistically significant difference among treatments. However, the Wilcoxon pairwise comparisons (Table 5.27) did not resulted in any significant differences between treatment pairs after correction.

Thus, despite a significant overall result, no specific treatment was statistically better than another,

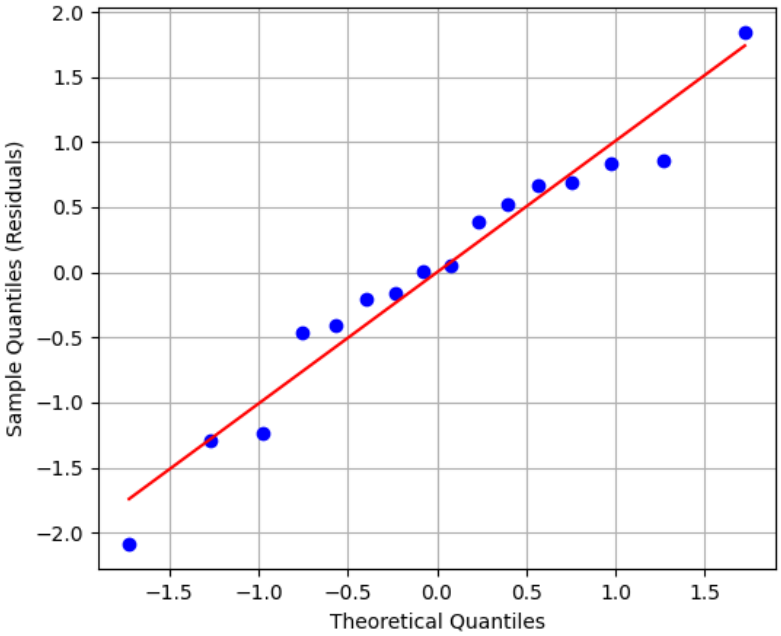

Figure 5.16: Normal Probability Plot of Residuals for Agent Architecture - Step 7

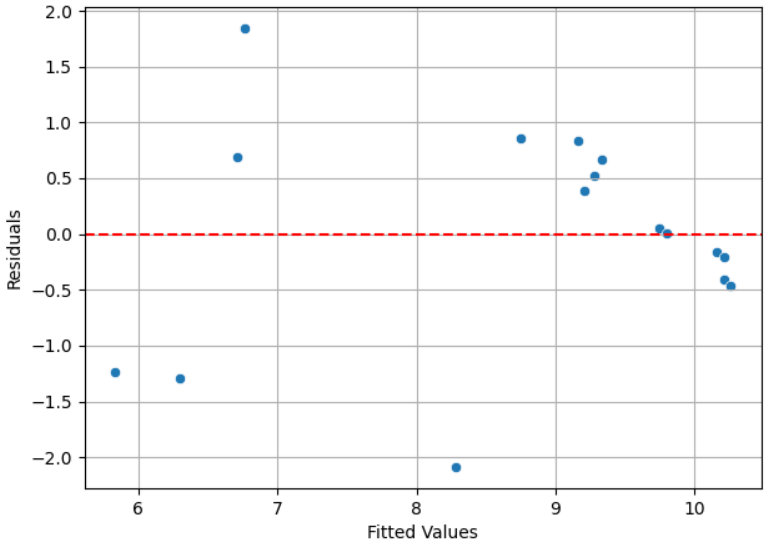

Figure 5.17: Residual Plot against Fitted Values for Agent Architecture - Step 7

though mean scores suggest an approximate ranking of $T4 > T3 > T2 > T1$.

Source of Variation	Chi-squared statistic	P-value
Treatment (Between Groups)	10.9412	0.0120

Table 5.26: Friedman's Calculation Results for Agent Architecture – Step 7.

Treatment1	Treatment2	W	p-unc	p-adj	$p - adj < 0.05$
T1	T2	0.00	0.1250	0.750	No
T1	T3	0.00	0.1250	0.750	No
T1	T4	0.00	0.1250	0.750	No
T2	T3	0.00	0.1340	0.750	No
T2	T4	0.00	0.1340	0.750	No
T3	T4	0.00	0.4533	0.750	No

Table 5.27: Pairwise Wilcoxon Results for Agent Architecture – Step 7.

Design: Deployment Diagram

Table 5.28 and Table 5.29 show the raw and summarized evaluation data.

Subject	Metric	T1	T2	T3	T4
S1	M1	10	10	10	10
	M2	10	10	10	8
	M3	10	10	10	10
	M4	9	10	10	10
	M5	10	10	10	10
S2	M1	10	10	10	10
	M2	10	10	10	10
	M3	10	10	10	10
	M4	10	10	10	10
	M5	10	10	10	10
S3	M1	9	7	10	10
	M2	10	5	10	10
	M3	10	8	10	9
	M4	10	4	10	10
	M5	10	7	10	10
S4	M1	10	10	10	8
	M2	10	10	10	10
	M3	10	10	10	10
	M4	10	10	10	10
	M5	10	10	10	10

Table 5.28: Raw Data for Deployment Diagram – Step 8.

Subject	T1	T2	T3	T4
S1	9.8	10	10	9.6
S2	10	10	10	10
S3	9.8	6.2	10	9.8
S4	10	10	10	9.6
Mean	9.9	9.05	10	9.75

Table 5.29: Summarized Data for Deployment Diagram – Step 8.

ANOVA Assumptions Check

Assumption checks (Figure 5.18 and Figure 5.19) identified normality and variance issues. The assumptions appear to be violated due to one or two influential data points that act as outliers, and the homoscedasticity is violated by the very large negative residual for one fitted value. Mauchly's test also indicated a violation of sphericity ($p = 0.0322$). Consequently, Friedman's ANOVA was applied.

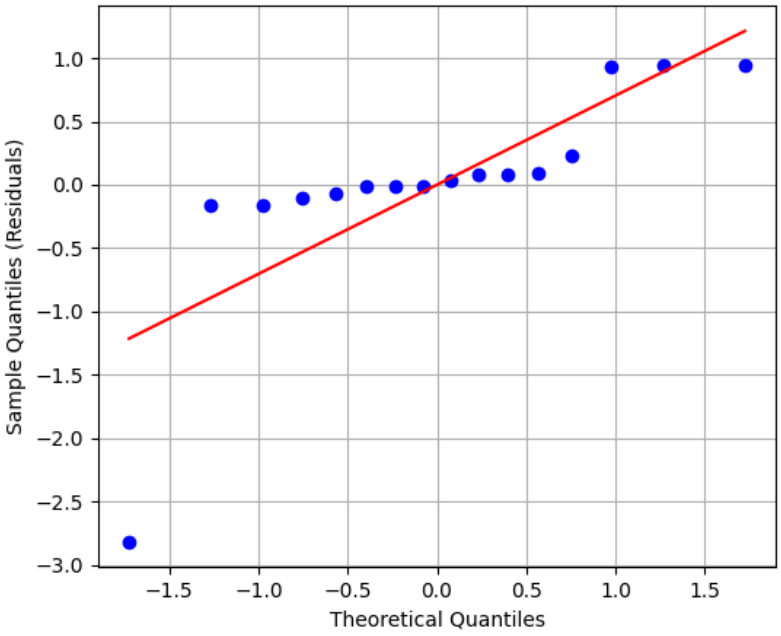

Figure 5.18: Normal Probability Plot of Residuals for Deployment Diagram - Step 8

Friedman's ANOVA Test

The Friedman's ANOVA test results (Table 5.30) did not indicate a significant difference between treatments. This implies that human and LLMs performed similarly in generating the Deployment Diagram.

Source of Variation	Chi-squared statistic	P-value
Treatment (Between Groups)	4.500	0.2123

Table 5.30: Friedman's ANOVA Calculation Results for Deployment Diagram – Step 8.

Code

Evaluation data is presented in Table 5.31 and Table 5.32.

ANOVA Assumptions Check

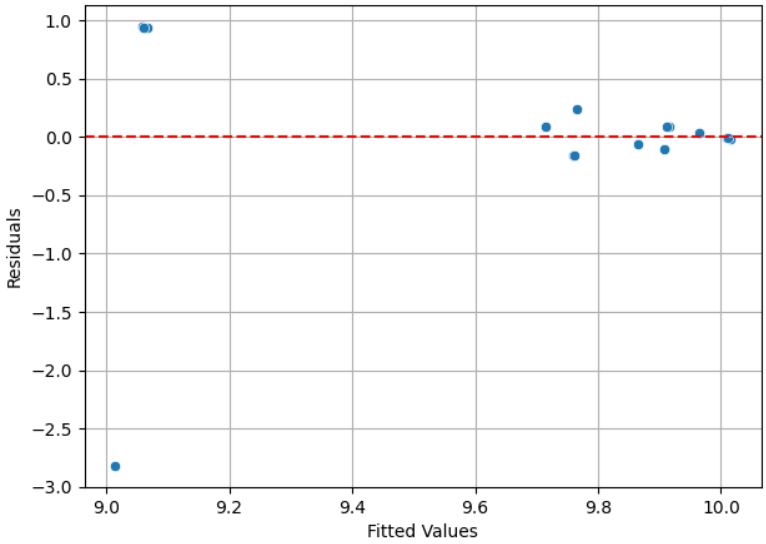

Figure 5.19: Residual Plot against Fitted Values for Deployment Diagram - Step 8

Subject	Metric	T1	T2	T3	T4
S1	M1	6	7	9	7
	M2	10	9	10	10
	M3	10	8	9	8
	M4	10	10	10	10
	M5	10	10	10	10
S2	M1	9	9	10	10
	M2	10	10	10	9
	M3	10	10	10	10
	M4	9	10	10	10
	M5	10	10	10	10
S3	M1	6	7	10	5
	M2	6	6	10	5
	M3	7	9	10	6
	M4	7	6	9	6
	M5	8	7	10	9
S4	M1	7	9	9	6
	M2	6	8	8	7
	M3	8	10	10	8
	M4	8	9	7	8
	M5	6	10	10	9

Table 5.31: Raw Data for Code – Step 9.

The assumption checks were satisfactory: the Q-Q plot (Figure 5.20) and the residuals plot (Figure 5.21) confirmed normality and variance homogeneity, and Mauchly's test ($p = 0.1504$) showed that sphericity was not violated.

RM ANOVA Test

Subject	T1	T2	T3	T4
S1	9.2	8.8	9.6	9.0
S2	9.6	9.8	10.0	9.8
S3	6.8	7.0	9.8	6.2
S4	7.0	9.2	8.8	7.6

Table 5.32: Summarized Data for Code – Step 9.

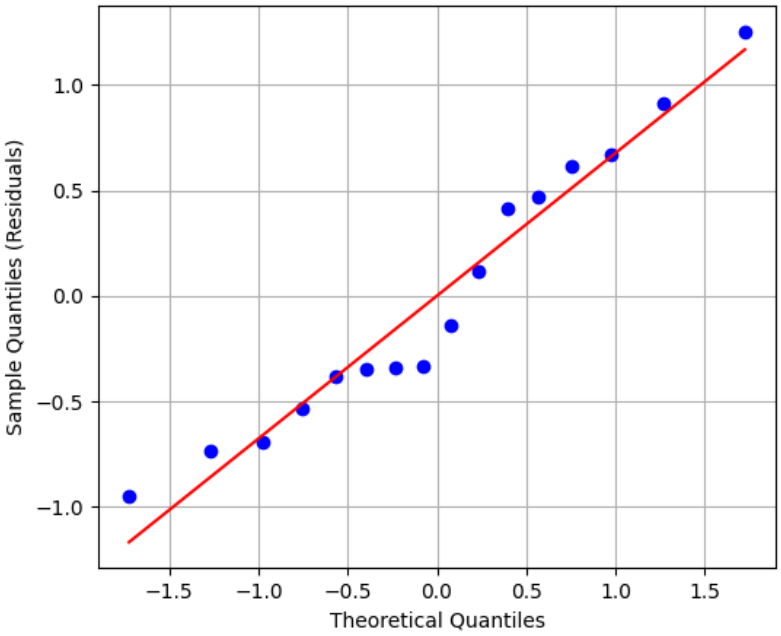

Figure 5.20: Normal Probability Plot of Residuals for Code - Step 9

RM ANOVA results (Table 5.33) showed no significant difference between treatments ($p = 0.1151$). This suggests that the quality of code produced by human and LLM-based approaches was statistically equivalent.

Source of Variation	SS	df	MS	F-value	P-value
Treatment (Between Groups)	5.2475	3	1.749	2.6053	0.1151
Error (Within Groups)	6.0425	9	0.6714		
Total		12			

Table 5.33: RM ANOVA Calculation Results for Code – Step 9.

5.4 Experiment Results (Presentation and Package)

This final phase involves documenting and summarizing the experimental results. Table 5.34 provides an overall view of all statistical outcomes across the nine steps of the LLM-based MaSE framework, including

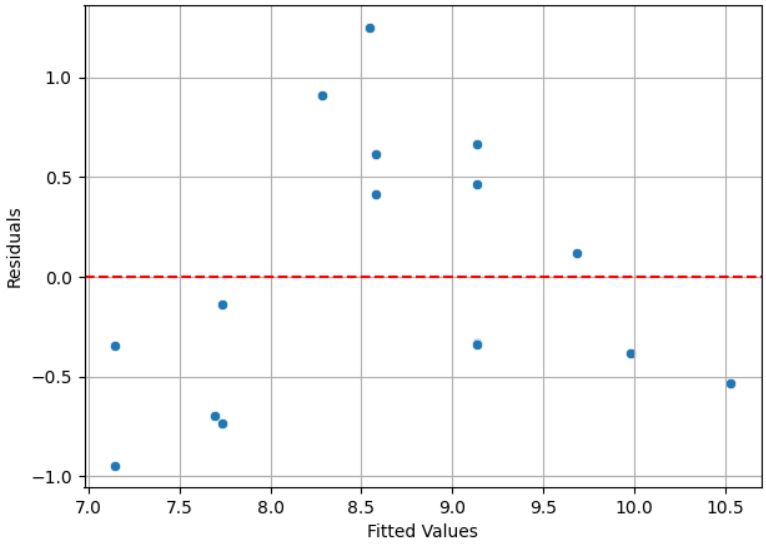

Figure 5.21: Residual Plot against Fitted Values for Code - Step 9

the results of ANOVA tests and pairwise comparisons.

Subject	Step	ANOVA (Significant Difference)	Pairwise Comparison (Significant Difference)	Pairs with Significant Difference
Analysis	Goal Hierarchy*	Yes (RM)	No	(T1, T3), (T1, T4) Before correction
	Sequence Diagram*	Yes (RM)	No	
	Role Model*	Yes (RM)	No	(T1, T3), (T1, T4) Before correction
	Concurrent Task Model	Yes (RM)	Yes	(T1, T3), (T1, T4)
Design	Agent Class Diagram*	Yes (Friedman's)	No	
	Communication Class Diagram	No (Friedman's)	–	All Equal
	Agent Architecture*	Yes (Friedman's)	No	
	Deployment Diagram	No (Friedman's)	–	All Equal
Code	Code	No (RM)	–	All Equal

Table 5.34: Overall Statistical Outcomes.

The table summarizes the following for each step:

- Whether the overall test (RM ANOVA or Friedman's ANOVA) found a significant difference across treatments.
- Whether the post-hoc pairwise comparisons identified any statistically significant differences.
- Which specific treatment pairs, if any, showed evidence of difference.

Five steps are marked with an asterisk (*), indicating that although ANOVA detected a significant difference between treatments, the follow-up pairwise comparisons failed to reflect the differences after correction.

In two of those cases, Goal Hierarchy and Role Model, the uncorrected p-values and other indicators such as Hedges' g and Bayes Factors suggest practically meaningful difference that might reach significance in studies with larger sample sizes. This outcome is consistent with the idea that ANOVA, as overall test with broader sensitivity, can detect overall group differences even when individual pairwise differences are harder to confirm with limited data.

Across all steps, the experimental results reveal two general trends: (1) In some cases, there is no statistically significant difference in performance between human and LLM-based treatments and (2) when significant differences are present, LLM performance, particularly that of Gemini (T3) and DeepSeek (T4), often exceeds that of the human approach (T1).

In terms of detailed interpretation:

- Strong statistical evidence shows that LLMs, especially Gemini and DeepSeek, outperformed human in the Concurrent Task Model step.
- Likely but not statistically confirmed LLM outperformance is observed in Goal Hierarchy and Role Model steps, where T3 and T4 had higher mean scores than T1, and uncorrected p-values suggest meaningful differences.
- In the Sequence Diagram, Agent Class Diagram, and Agent Architecture steps, although ANOVA detected significant differences, pairwise comparisons did not confirm any clear treatment superiority, possibly due to limited statistical power.
- In Communication Class Diagram, Deployment Diagram, and Code steps, there is no significant difference, indicating that LLMs and human developers performed equally.

From a broader perspective, in the Analysis phase of MaSE, all LLMs demonstrate similar performance, with human typically lagged behind, and Gemini leading slightly among models. In the Design phase, two steps showed no significant differences across treatments, and the same conclusion holds for the Code step. Overall, based on the sub-experiments done for each step of the MaSE methodology and the statistical analysis applied, it is evident that LLMs can perform as well as or better than human in designing and developing multi-agent systems. While performance across LLMs was generally consistent, Gemini showed a slight advantage in multiple steps.

5.5 Threats to Validity

The main threat to the validity of this experiment is that it is performed on a single, small project from a know domain. The results of this experiment may be valid just for this particular case, and the generalization of the findings relies on conducting more experiments on a broader set of MAS development projects, larger in scale and from diverse domains.

Also, we conducted the experiment with a relatively small dataset, including only 4 subjects (raters). This small sample size limits the statistical power, especially for post-hoc pairwise comparison, failing to detect real differences between treatments. This is evident in several steps where the ANOVA indicated significant differences, but pairwise comparisons did not confirm them due to insufficient power after correction.

Another limitation is the LLM-as-a-judge approach. While increasingly adopted, this introduces the possibility of alignment bias, where LLMs may favor outputs similar to their own generation style.

Additionally, the experiment relied on a specific set of LLMs; even different versions or sizes within the same model families could produce varying results.

To obtain more confident and generalizable results, future experiments should:

- Include more diverse and complex MAS projects.
- Involve a larger and more balanced set of raters (including both human experts and LLMs).
- Validate LLM-judging consistency with expert human reviewers to ensure robustness.
- Incorporate a wider variety of LLMs, including alternate versions within the same families and models from other leading families.

This study evaluates the quality of artefacts produced during the analysis, design, and implementation stages of MAS development, focusing on structure, traceability, and methodological correctness. Consequently, runtime behavior, performance, maintainability, and developer experience were not evaluated and are considered outside the scope of this thesis.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis introduced a novel framework for leveraging Large Language Models (LLMs) in the field of software engineering, with specific emphasis on the development of Multi-Agent Systems (MAS). The proposed framework integrates LLMs across the entire software development lifecycle, from requirements analysis to system design and code generation, guided by prompt engineering techniques.

At its core, the framework employs structured and predefined prompts that guide the LLM in generating MAS artefacts in alignment with the MaSE (Multiagent Systems Engineering) methodology and implementing them using the JADE (Java Agent Development) framework.

Empirical evaluation, conducted through a case study, demonstrated the ability of LLMs to generate modular, traceable, and high-quality software artefacts that are often similar to human-generated outputs and sometimes better. Additionally, an experiment was designed and conducted to compare human developers with LLMs using structured metrics and statistical analysis. The experiment results revealed that LLMs can perform MAS development tasks with comparable, and at times superior, results.

Moreover, the conducted experiment, supported by quantifying measures and statistical analysis, contributes to the evaluation of LLMs in software engineering. It introduced an evaluation strategy for assessing LLM-generated outputs by combining human judgment with LLM-as-a-judge mechanisms.

However, several threats to validity must be acknowledged. First, the experiment was conducted on a single, relatively small MAS project within a familiar domain. This raises questions about the generalizability of the findings. Larger and more diverse case studies are needed to validate the framework's robustness across contexts. Second, the experiment involved only four raters, which limited its statistical power. This made

post-hoc comparisons unable to detect real differences between treatments due to low sample size. This limitation was reflected in instances where ANOVA reported significant results but pairwise comparisons did not show any significant differences. Third, while the use of LLMs as judges is a growing trend, it introduces a risk of alignment bias. This means LLMs may favor outputs that mirror their own generation styles. Lastly, our evaluation was conducted using a specific set of LLMs, and results may vary with other models, versions, or parameter settings.

6.2 Future work

Based on the results and limitations of this thesis, we suggest several directions for future work:

1. Scaling to more diverse and complex MAS projects

   To improve the validity and generalizability of results, the proposed framework should be applied and tested across a broader range of MAS development projects. This would help determine its effectiveness for larger, more complex systems and across different domains.

2. Expanding and balancing the evaluator set

   Increasing the number of raters, including both human experts and LLMs, will enhance the reliability of evaluation results. A sufficiently large sample size is essential for making meaningful comparisons and uncovering performance differences between humans and LLM-generated outputs.

3. Validating the LLM-as-a-judge consistency

   While LLMs offer useful evaluation capabilities, further research should investigate their potential alignment bias and consistency. This can be achieved by comparing LLM-based assessments with evaluations conducted by human experts.

4. Integrating a user feedback mechanism

   Future versions of the supporting tool should integrate human feedback at every step of the development process. This includes mechanisms for refining and adjusting prompts and generated artefacts, as well as enabling users to navigate both forward and backward through the development process .

5. Developing a repository of best practices for MAS development

   Creating a repository of best practices for MAS development, based on human feedback during the development process, can serve as a valuable knowledge base for LLMs. This would enhance the models' domain understanding and lead to higher-quality outputs.

6. Evaluating with diverse LLMs from various LLM families

   Future experiments should include a wider range of LLMs, models of varying sizes and from different model families. Comparing outputs from different models and even different versions within the same family will help us to better understand how LLM characteristics influence the quality and reliability of generated software artefacts.

   In conclusion, this research provides a strong starting point for using LLMs in MAS development, while also showing that more work is needed to improve and validate the approach. By tackling the current limitations and exploring the suggested future directions, developers can better use LLMs to create smart, reliable, and collaborative multi-agent systems.

Bibliography

[1] University of Calgary Faculty of Graduate Studies. Guidelines for generative ai use in graduate studies. `https://perma.cc/782F-QH65`, April 2025.

[2] Michael J. Wooldridge. *An introduction to multiagent systems*. Wiley, Chichester, 2. ed., repr edition, 2012.

[3] Scott A. Deloach, Mark F. Wood, and Clint H. Sparkman. MULTIAGENT SYSTEMS ENGINEERING. *Int. J. Soft. Eng. Knowl. Eng.*, 11(03):231–258, June 2001.

[4] Kingsley Okoye and Samira Hosseini. *R Programming: Statistical Data Analysis in Research*. Springer Nature, July 2024. Google-Books-ID: Wd8SEQAAQBAJ.

[5] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. A Survey on Large Language Models for Software Engineering, December 2023. arXiv:2312.15223 [cs].

[6] Enrique Dehaerne, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. Code Generation Using Machine Learning: A Systematic Review. *IEEE Access*, 10:82434–82455, 2022.

[7] I. Ozkaya. Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications. *IEEE Software*, 40:4–8, 2023.

[8] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology*, 33:1–79, 2024.

[9] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53, Melbourne, Australia, May 2023. IEEE.

[10] Tianyou Chang, Shizhan Chen, Guodong Fan, and Zhiyong Feng. A Self-Iteration Code Generation Method Based on Large Language Models. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 275–281, December 2023. ISSN: 2690-5965.

[11] Dipti Srinivasan, Lakhmi C. Jain, and Janusz Kacprzyk, editors. *Innovations in Multi-Agent Systems and Applications - 1*, volume 310 of *Studies in Computational Intelligence*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[12] Scott A. DeLoach. The MaSE Methodology. In Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli, editors, *Methodologies and Software Engineering for Agent Systems*, volume 11, pages 107–125. Kluwer Academic Publishers, Boston, 2004. Series Title: Multiagent Systems, Artificial Societies, and Simulated Organizations.

[13] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Developing Multi-agent Systems with JADE. In Cristiano Castelfranchi and Yves Lespérance, editors, *Intelligent Agents VII Agent Theories Architectures and Languages*, pages 89–103, Berlin, Heidelberg, 2001. Springer.

[14] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review, September 2024. arXiv:2310.14735 [cs].

[15] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. Prompt Engineering or Fine Tuning: An Empirical Assessment of Large Language Models in Automated Software Engineering Tasks, October 2023. arXiv:2310.10508 [cs].

[16] Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models, June 2024. arXiv:2405.06211 [cs].

[17] Christof Ebert and Panos Louridas. Generative AI for Software Practitioners. *IEEE Softw.*, 40(4):30–38, July 2023.

[18] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, March 2022. arXiv:2203.02155 [cs].

[19] William Cain. Prompting Change: Exploring Prompt Engineering in Large Language Model AI and Its Potential to Transform Education. *TechTrends*, 68(1):47–57, January 2024.

[20] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. Prompt Engineering in Large Language Models. In I. Jeena Jacob, Selwyn Piramuthu, and Przemyslaw Falkowski-Gilski, editors, *Data Intelligence and Cognitive Informatics*, pages 387–402, Singapore, 2024. Springer Nature.

[21] Shubham Vatsal and Harsh Dubey. A Survey of Prompt Engineering Methods in Large Language Models for Different NLP Tasks, July 2024. arXiv:2407.12994 [cs].

[22] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT, February 2023. arXiv:2302.11382 [cs].

[23] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design. In Anh Nguyen-Duc, Pekka Abrahamsson, and Foutse Khomh, editors, *Generative AI for Effective Software Development*, pages 71–108. Springer Nature Switzerland, Cham, 2024.

[24] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, July 2021. Publication Title: arXiv e-prints ADS Bibcode: 2021arXiv210703374C.

[25] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A Generative Model for Code Infilling and Synthesis, April 2023. arXiv:2204.05999 [cs].

[26] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis, February 2023. arXiv:2203.13474 [cs].

[27] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open Foundation Models for Code, January 2024. arXiv:2308.12950 [cs].

[28] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks Are All You Need, October 2023. arXiv:2306.11644 [cs].

[29] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, December 2022. Publisher: American Association for the Advancement of Science.

[30] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. Discovering the Syntax and Strategies of Natural Language Programming with Generative Language Models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, pages 1–19, New York, NY, USA, April 2022. Association for Computing Machinery.

[31] Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. On the Effectiveness of Large Language Models in Domain-Specific Code Generation. *ACM Trans. Softw. Eng. Methodol.*, page 3697012, October 2024.

[32] Leon Chemnitz, David Reichenbach, Hani Aldebes, Mariam Naveed, Krishna Narasimhan, and Mira Mezini. Towards Code Generation from BDD Test Case Specifications: A Vision. In *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*, pages 139–144, May 2023.

[33] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-Planning Code Generation with Large Language Models. *ACM Trans. Softw. Eng. Methodol.*, 33(7):182:1–182:30, September 2024.

[34] Fernando Vallecillos Ruiz. Agent-Driven Automatic Software Improvement. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, pages 470–475, New York, NY, USA, June 2024. Association for Computing Machinery.

[35] Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawendé F. Bissyandé. CodeAgent: Autonomous Communicative Agents for Code Review. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11279–11313, Miami, Florida, USA, November 2024. Association for Computational Linguistics.

[36] Zeeshan Rasheed, Malik Abdul Sami, Kai-Kristian Kemell, Muhammad Waseem, Mika Saari, Kari Systä, and Pekka Abrahamsson. CodePori: Large-Scale System for Autonomous Software Development Using Multi-Agent Technology, September 2024. arXiv:2402.01411 [cs].

[37] Malik Abdul Sami, Muhammad Waseem, Zeeshan Rasheed, Mika Saari, Kari Systä, and Pekka Abrahamsson. Experimenting with Multi-Agent Software Development: Towards a Unified Platform, June 2024. arXiv:2406.05381 [cs].

[38] Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework, November 2024. arXiv:2308.00352 [cs].

[39] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative Agents for Software Development, June 2024. arXiv:2307.07924 [cs].

[40] Simiao Zhang, Jiaping Wang, Guoliang Dong, Jun Sun, Yueling Zhang, and Geguang Pu. Experimenting a New Programming Practice with LLMs, January 2024. arXiv:2401.01062 [cs].

[41] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T. Nguyen, and Riccardo Rubei. On the use of large language models in model-driven engineering. *Softw Syst Model*, January 2025.

[42] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2024.

[43] Alessio Ferrari, Sallam Abualhaija, and Chetan Arora. Model Generation with LLMs: From Requirements to UML Sequence Diagrams. In *2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW)*, pages 291–300, June 2024. ISSN: 2770-6834.

[44] Abdel-Halim Hafez Elamy and Behrouz Far. On the evaluation of agent-oriented software engineering methodologies: A statistical approach. In Manuel Kolp, Brian Henderson-Sellers, Haralambos Mouratidis, Alessandro Garcia, Aditya K. Ghose, and Paolo Bresciani, editors, *Agent-Oriented Information Systems IV*, pages 105–122. Springer.

[45] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. 36:46595–46623.

Appendix A

Prompts for LLM-as-a-judge Assessment

The phase results files were uploaded and the following instructions were given to the LLM for assessment purposes.

Prompt for Analysis Phase Assessment

In the attachment file (`Analysis.txt`) are the steps of the MaSE analysis phase for multi-agent system development, documented in textual format by a software engineer.

The process starts with a pre-process step which is the requirement refinement based on the initial requirements. In the following process, the aim is to develop a Shopping System MAS application with the following initial requirements:

> "There are three different roles, Clients who buy products, Suppliers who sell products, and a Broker, a special role that introduces Clients to Suppliers. All negotiations happen directly between Client and Supplier afterwards.
>
> The main workflow of this system is that every Supplier registers to the Broker, sending all its products' information to the Broker. Clients can send a list of required products to the Broker. The Broker searches the list information sent by Suppliers; if there exists a Supplier that matches the request (provides the requested products), the Broker will send the Supplier's name to the Client. Then the Client and Supplier make a connection

and negotiate for the products. After the bargain is finished, if the product(s) were sold to the Client, the Supplier needs to update its product information to the Broker."

Starting from step 1, mark each step based on the following criteria (each out of 10):

1. **Completeness (Consistency):** To what extent does the model or code align with the system requirements and previously generated models?
2. **Correctness:** To what extent does the model or code correctly reflect the structure and behavior of the system?
3. **Adherence to the standard (syntax, semantic):** To what extent does the model or code adhere to the expected syntax and maintain semantic correctness?
4. **Understandability:** To what extent is the model or code clear and without redundancies?
5. **Terminology alignment:** To what extent does the terminology used in the model or code align with the system requirements and previously generated models?

Here is the refined requirement list extracted by the engineer as a pre-process step:

{`refined requirement list`}

Note: Mark based on the MaSE methodology step definitions and rules.

Prompt for Design Phase Assessment

In the attachment file (`Design.txt`) are the steps of the MaSE design phase for multi-agent system development, documented in textual format by a software engineer.

Starting from step 1, mark each step based on the following criteria (each out of 10):

1. **Completeness (Consistency):** To what extent does the model or code align with the system requirements and previously generated models?
2. **Correctness:** To what extent does the model or code correctly reflect the structure and behavior of the system?
3. **Adherence to the standard (syntax, semantic):** To what extent does the model or code adhere to the expected syntax and maintain semantic correctness?
4. **Understandability:** To what extent is the model or code clear and without redundancies?

5. **Terminology alignment:** To what extent does the terminology used in the model or code align with the system requirements and previously generated models?

Note: Mark based on the MaSE methodology step definitions and rules.

Prompt for Code Assessment

In the attachment file (`Code.txt`) is the implementation code for the designed multi-agent system, developed by the software engineer.

Please assess the code based on the following criteria (each out of 10):

1. **Completeness (Consistency):** To what extent does the code align with the system requirements and previously generated models?
2. **Correctness:** To what extent does the code correctly reflect the structure and behavior of the system?
3. **Adherence to the standard (syntax, semantic):** To what extent does the code adhere to the expected syntax and maintain semantic correctness?
4. **Understandability:** To what extent is the code clear and without redundancies?
5. **Terminology alignment:** To what extent does the terminology used in the code align with the system requirements and previously generated models?

Assessment Tips:

- Check that all agents are implemented based on the agent class diagram.
- Check that all agents' behaviours are implemented correctly and completely.
- Check that all internal classes (with their attributes and operations) are implemented in the corresponding agent class.
- Check that all containers, plus the main container, are implemented and instantiating their agents based on the deployment diagram.
- The code should be runnable without any syntax or semantic errors.