NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

TIAGO GONÇALO BANDOS RODRIGUES

BSc in Informatics Engineering

GENERATIVE AI FOR CLASS DIAGRAM MODELING FROM SOFTWARE REQUIREMENTS

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2025

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

GENERATIVE AI FOR CLASS DIAGRAM MODELING FROM SOFTWARE REQUIREMENTS

TIAGO GONÇALO BANDOS RODRIGUES

BSc in Informatics Engineering

Advisers: João Araújo
Associate Professor, NOVA University Lisbon
Ana Moreira
Full professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2025

Generative AI for Class Diagram Modeling from software requirements

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.2.5) [1].

Acknowledgements

I would like to express my deepest gratitude to my adviser, Professor João Araújo, and the co-adviser, Professor Ana Moreira, for their invaluable support and guidance throughout this research journey. Their expertise, encouragement, and insightful feedback have been instrumental in shaping this thesis. I also would like to thank Professor João M.Lourenço for providing a template for this thesis [1]. I want to extend my thanks to all Department of Informatics, specifically to the professor João Leitão for giving me access to the clusters of the Department.

I am thankful for my colleagues and friends I made during this five years for their support during this thesis, the Master's program, and the Bachelor's degree. Without them it would have been impossible to finish this journey.

I would also like to extend my appreciation to both my parents, that were next to me every step of the way, helping me to overcome difficulties and never letting me give up, and my brother for discussing with me ideas about generative AI and for all feedback given. Last but not least, I want to thank Filipa Ferraz for all the patient during all this months, for listining me talking about AI all the time and for all the support given.

I am grateful to have such amazing people in my life, and I look forward to carrying the lessons learned and the support received into my future projects.

Additionally, I acknowledged the use of Generative AI tools, such as ChatGPT and Claude, to help me improve the quality of my written english by fixing grammar and spelling mistakes and for help me better structuring the text through this thesis, to enhance the clarity, coherence, and overall quality of the text. All content was created by me, and I carefully reviewed and edited the AI-generated suggestions to ensure accuracy and alignment with my intended meaning.

”

“Generative models are changing the way we think about machine intelligence and creativity, and have the potential to transform industries from media to finance to healthcare.”

— Oriol Vinyals,
(Research Scientist at Google.)

Abstract

Software modeling demands a good understanding of the solution domain, as well as requirements specification. The resulting models serve as blueprints for the hardware and software requirements, outlining system architecture, implementation phase, and the overall system structure. To automate the creation of such models various methods have been proposed over the years, including approaches based on model-driven development, linguistic rules and patterns, rule-based systems, and statistical or machine learning techniques. More recently, Large Language Models (LLMs) have gained prominence. While LLMs often show promising results they commonly lack sufficient contextual awareness, an essential factor in accurately transforming requirement specifications into models such as class diagrams, and output often is inconsistent. This dissertation addresses this limitation by introducing an approach that combines prompt engineering techniques, a Retrieval-Augmented Generation system, and a self-refinement mechanism. The proposed pipeline provides an automated process for transforming natural language descriptions into UML class diagrams. It demonstrates the potential of Generative AI to enhance efficiency and reduce reliance on manual effort in the creation of models. The approach was evaluated using a combination of qualitative and quantitative methods, including user studies and performance metrics. Results indicate that AI-generated diagrams consistently outperform human-created diagrams in syntactic, semantic, and pragmatic quality, with over 65% of the thirty evaluators unable to distinguish AI outputs from human ones and with over 73% of evaluators preferring the diagram generated by our system over the human diagram. While pragmatic quality remains the most challenging dimension, findings confirm that Generative AI can produce professional-quality diagrams that enhance communication among stakeholders and support early design decisions.

Keywords: LLM, Requirement modeling, Retrieval-Augmented Generation, UML diagrams, GAI

Resumo

A modelação de software requer uma boa compreensão do domínio da solução e da respetiva especificação de requisitos. Os modelos resultantes servem como planos para a implementação, definindo a arquitetura do sistema, os requisitos de hardware e software, e a estrutura global do sistema. Ao longo dos anos, foram propostos diversos métodos para automatizar a criação destes modelos, incluindo abordagens baseadas em desenvolvimento orientado por modelos, regras e padrões linguísticos, sistemas baseados em regras e técnicas estatísticas ou de aprendizagem automática. Mais recentemente, os *Large Language Models (LLMs)* ganharam destaque. Embora os LLMs apresentem frequentemente resultados promissores, revelam frequentemente falta de consciência contextual, um fator essencial para transformar com precisão especificações de requisitos em modelos como diagramas de classes. Esta dissertação procura colmatar esta limitação através da introdução de uma abordagem que combina técnicas de engenharia de prompts, um sistema de Recuperação Aumentada (RAG) e um mecanismo de auto-aperfeiçoamento. O *pipeline* proposto fornece um processo automatizado para transformar descrições em linguagem natural em diagramas de classes UML. Demonstra ainda o potencial da Inteligência Artificial Generativa para aumentar a eficiência e reduzir a dependência do esforço manual na criação de modelos. A abordagem foi avaliada através de uma combinação de métodos qualitativos e quantitativos, incluindo estudos com especialistas e métricas de desempenho. Os resultados indicam que os diagramas gerados por IA superam consistentemente os diagramas criados por humanos nas dimensões sintática, semântica e pragmática, com mais de 65% dos trinta avaliadores incapazes de distinguir os diagramas criados com IA dos criados por humanos. Embora a qualidade pragmática continue a ser a dimensão mais desafiante, os resultados confirmam que a IA generativa pode produzir diagramas de qualidade profissional que melhoram a comunicação entre as partes interessadas e apoiam a tomada de decisões nas fases iniciais de design.

Palavras-chave: LLM, Modelação de requisitos, Retrieval-Augmented Generation, diagramas UML, GAI

Contents

List of Figures

Listings

1

Introduction

This research was motivated by recent advancements in General Artificial Intelligence (GAI) and the ongoing need to automate requirement modeling. Automating this process has the potential to significantly enhance communication with stakeholders by producing clearer and more comprehensive diagrams, while also reducing the time spent on software modeling and thereby increasing overall efficiency. This introductory chapter provides an overview of the context and motivation of this dissertation, defines the problem statement and the associated challenges, and presents the objectives of this thesis preparation along with the expected contributions.

1.1 Context and Motivation

Software modeling is an essential process in software engineering, that involves creating abstract representations of the most significant aspects of the software, facilitating the identification of potential issues early, improving design consistency, and providing a blueprint for implementation and maintenance. Traditionally, this process is carried out manually by software engineers, which can be time-consuming, heavily reliant on human expertise, and prone to human error [2].

Over the past two years, several approaches with a combination of LLM and rule-based system [3] or machine-learning systems [4] have been developed to automate the process of generating design models from requirements, with the goal of improving efficiency and minimizing dependence on manual effort. Although these approaches have shown promising results, they still present limitation in terms of accurately identifying complex relationships, often producing hallucinated output [5]. Additionally, issues such as ambiguity in natural language [6] and limited domain-specific knowledge [7] limit the adaptability and accuracy of these techniques in practical real-world scenarios.

1.2 Problem statement

We defined the problem statement as the lack of knowledge domain, and the inconsistency of outputs presented by LLM, which lead to incomplete and inaccurate requirement models. The problem statement was based on the findings of the systematic mapping study we conducted. The discussion that follows identifies and justifies the main problem that comprises the problem statement, the challenges that arise from these problems, and a propose solution to address them.

Problem: The lack of domain-specific knowledge and the inconsistency of outputs presented by LLM presented by LLM lead to incomplete and inaccurate requirement models.

Our systematic mapping study highlights key challenges impacting the completeness and accuracy of requirement models generated. Main issues include ambiguity, vagueness, and inconsistency in the input requirements, which affect the model's ability to produce reliable output. [8]. To mitigate these issues, clear and well-structured input specifications, combined with iterative prompting, are critical measures to enhance the accuracy and relevance of generated models [6]. In this context, prompt engineering plays a crucial role in addressing input-related challenges, helping to guide models toward more precise and contextually appropriate results. Beyond input quality, LLMs exhibit problems in identifying complex relationships and attributes, which often leads to lower correctness rates in the generated models [9]. This limitation can be associated with the fact that the models lack domain-specific knowledge, resulting in outputs that are not only incomplete but sometimes hallucinated or contextually incorrect. Furthermore, the restricted scope of an LLM's training dataset limits its ability to generalize effectively across diverse or specialized domains. The identification of these difficulties led to the definitions of the following challenges:

Challenge 1: Define the best prompt to generate models from requirements.

The performance of LLMs depends heavily on the quality of the prompts, as noted in the systematic mapping study. Techniques like zero-shot, few-shot and chain-of-thought (CoT) prompting have shown potential to guide LLMs in generating more accurate and relevant outputs. It is important the prompts are clear, detail and domain-specific to ensure accurate and complete models.

Challenge 2: Defining the data relevant to improve the domain knowledge.

LLMs often struggle with domain-specific tasks due to the general nature of their training data. This can lead to hallucinations and incomplete outputs, specially for complex or specialized requirements. Incorporating domain-specific knowledge is a critical step in addressing this limitation. This can be done by implementing a Retrieval Augmented Generation (RAG) system. Identifying and collect relevant data for augmenting LLMs can improve their contextual understanding and ability to generate accurate models.

Challenge 3: Avoid ambiguity, vagueness, and incompleteness in the requirements.

Natural language requirements are naturally ambiguous, vague or inconsistent, which

can lead to errors in automated model generation. This challenge ties on the analysing the requirements in order to mitigate the ambiguity and vagueness of the requirements by refining it. Its important to have clear and consistent initial requirements before automating the creation of the model.

1.3 Objectives

The main objective of this thesis is to propose an approach to automate the design modelling from requirements. To achieve our purpose, we propose a automated pipeline that combines prompt engineering techniques with a RAG system to enhance the domain-specific knowledge of LLMs. To reduce the inconsistency of the outputs, we implemented a self refinement loop that analyses and refines the UML class Diagram initial generated to mitigate ambiguity, vagueness, and incompleteness.

1.4 Contributions

This thesis provides many contributions to differing fields, which are as follows:

1. A systematic mapping study of different techniques used to automate design modeling from requirements, providing a structured overview of current approaches and identifying gaps in existing research.

2. A structured prompt template to translate system descriptions into UML class diagrams and other UML diagrams. This template addresses the ambiguity, vagueness, and incompleteness often found in real-world requirements, resulting in more reliable outputs from generative models.

3. A simple Retrieval-Augmented Generation (RAG) system to enhance the prompts provided to Generative AI systems in the context of UML diagram generation.

4. A self-refinement cycle for improving UML diagrams and mitigating inconsistent output quality.

5. A novel automatic evaluation framework for UML class diagrams based on three quality dimensions: syntactic, semantic, and pragmatic quality.

6. Findings that can inform and contribute to ongoing debates on the role of AI in software modeling.

1.5 Document structure

The remainder of this document includes three chapters and is structured as follows:

- Chapter 2, Background, provides an overview of software modeling, large language models, and the principles underlying systematic mapping studies, along with an introduction to their methodology.
- Chapter 3, State of the art, presents the conducted systematic mapping study and a discussion of its results, and provides an overview of the current state.
- Chapter 4, Experiments with Generative AI techniques and tools, presents experiments with different Generative AI techniques and tools to validate possible solutions for the main problems found in the previous Chapters.
- Chapter 5, An approach to automate the creation of class diagrams, describes the proposed approach to automate the creation of UML class diagrams from natural language requirements, detailing the architecture, components, and workflow of the system.
- Chapter 6, Evaluation, presents the evaluation procedure of the proposed approach, including the questionnaire and G-evaluation results, and discusses the implications of the findings.
- Chapter 7, Conclusions and Future Work, summarizes the main contributions of the thesis, discusses its limitations, and outlines potential directions for future research.

2

Background

The present research work aims to propose an approach to improve requirements modeling by leveraging Generative AI (GAI) techniques to automate the generation of UML diagrams during the software design phase. The purpose of this chapter is to briefly introduce the main concepts in the field of software modeling that serve as the foundation for this thesis, namely tools, techniques, and approaches for translating requirements into design models, with a particular focus on UML class diagrams and their key components (classes, attributes, operations, and relationships). Additionally, we provide an overview of Generative AI and Retrieval-Augmented Generation (RAG), which form the core technologies employed in this research. This chapter also discusses related work and systematic methodologies, such as systematic mapping study, conducted to identify gaps and opportunities addressed in this thesis.

2.1 Software Modeling

Modelling is a fundamental practice in science and engineering, used to create abstract representations of a system at varying levels of precision and detail. These models serve as analytical tools, enabling a deeper understanding of the system's structure, behaviour, and constraints throughout its development process [10] This thesis will use Unified Modelling Language (UML) to produce design models from requirements.

Unified modelling language was developed to provide a standardized graphical language and notation for describing object-oriented models. UML notation is one of the most use language for describing software requirements, analysis and design models. The first time UML was introduced was by Grady Booch, James Rumbaugh, and Ivar Jacobson in 1999 [11] and after was officially adopted in January 1997 by OMG. In the OMG'S view, "modeling is the designing of software applications before coding.".

Unified Modeling Language (UML) is not only used for object-oriented design but also for modeling and functional aspects of systems [12]. The current standard version, UML 2.5.1, continues to evolve and supports various diagrams for application development, including use case diagrams, class diagrams, object diagrams, communication diagrams,

sequence diagrams, state machine diagrams, activity diagrams, composite structure diagrams, and deployment diagrams. In this thesis, we focus on class diagrams for design modeling. Class diagrams are the most widely used UML diagrams [13], providing a visual representation of a system's structure by illustrating classes, their attributes, methods, and the relationships among them. In addition to aiding system visualization, class diagrams also serve as a documentation tool and design blueprint, ensuring consistency and alignment with the system's intended functionality.

2.1.1 Benefits for developers and stakeholders

Class diagrams present benefits for both developers and stakeholders, as it acts as a universal language, reducing the gap between technical and non-technical stakeholders. By providing a clear and concise visual representation, it guarantees that everyone involved is in understanding. Also, it helps identifying potential design issues early in the development process because of the clear and concise visualization of the design. Additionally, class diagrams help ensure quality by for example ensuring that the system respects certain design principles or patterns, which can further improve the quality and maintenance of the software [14].

2.1.2 Core components of UML class diagrams

In UML Class diagram, classes are represented as rectangular boxes, with their attributes and operations detailed inside. The relationships between classes are represented using different types of connectors, which demonstrate structural and behavioural associations within the system. There are four primary types of relationships: associations, whole/part relationships, and generalization/specialization relationships and dependencies. An association represents a static structural relationship between two or more classes. It defines how instances of one class are related to instances of another class [10]. Associations can have multiplicity, which specifies how many instances of one class can be associated with instance of another class, (see Figure 2.1 a). The multiplicity of an association can be optional (0..1), exactly one (1), zero or more (*), numerically specified (m..n), or one or more (1..*)

Whole/part relationships define hierarchical relationships between objects. There are two types, Aggregation and Composition. Aggregation represents a weak relationship between the whole and its parts, where the part can exist independently of the whole. It is portrayed with a hollow diamond at the end of the association. In contrast, composition represents a strong relationship where the part cannot exist independently of the whole. It is presented as a filled diamond at the end of the association (see Figure 2.1 b). Generalization/Specialization Hierarchy is an inheritance relationship, where a subclass inherits properties from a superclass. This allows for the reuse of common attributes and behaviors. It is represented by an arrow joining the subclass (child) to the superclass (parent) (see Figure 2.1 c) A dependency relationship is often used to show how packages

are related, and represents a weaker relationship where one class depends on another for its operation

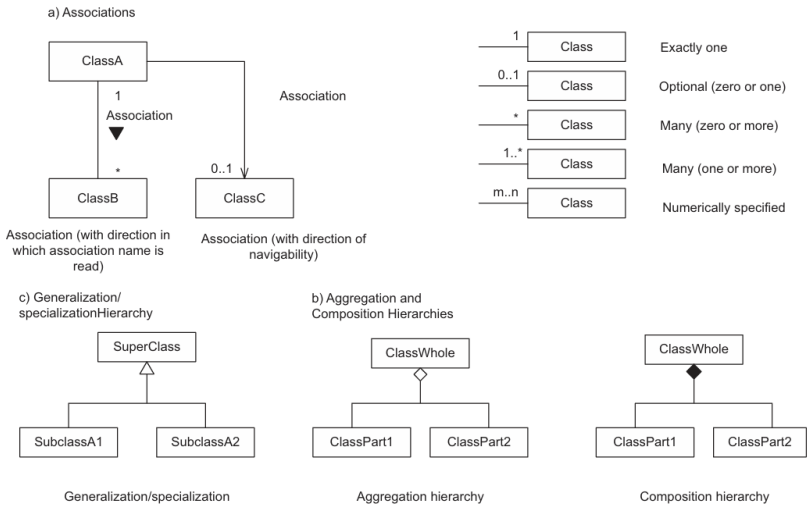


Figure 2.1: UML notation for relationships on a class diagram [10]

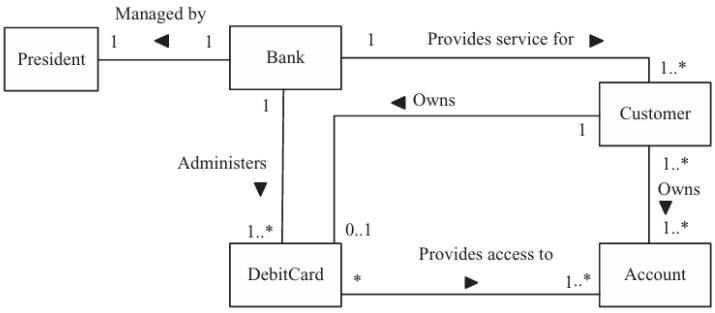


Figure 2.2: Example of relationships on a class diagram [10]

2.1.3 UML design class diagrams

The design process involves several key steps to ensure that the diagram accurately models the system's domain and behavior. By following a structured approach, designers can systematically identify the key components, define their relationships, and establish clear interactions within the system. The following steps outline a systematic process for designing a UML class diagram:

1. **Requirements Analysis:** understand the system's requirements by identifying the main entities or classes that will be part of the system.

2. **Identifying Relationships:** identify how the class will interact with the others by determining the type of relationships.

3. **Define Multiplicity:** determine the multiplicity of the association, i.e. specify how many instances of one class may relate to a single instance of another class [10].
4. **Add attributes and methods:** for each class list attribute (properties, fields) and methods (functions), it will have.
5. **Draw the diagram:** begin by creating class boxes with their names. Next, add attributes and methods to define each class's structure and behavior. Finally, establish relationships between classes, ensuring proper associations, generalizations, and dependencies, while specifying multiplicity where necessary.
6. **Review and refine:** review the diagram for accuracy, completeness, and consistency. Refine it as needed to ensure clarity and correctness.

There exists many different tools and software used to representing class diagrams, the most popular are Microsoft Visio [15], StarUML [16], PlantUML [17], and Lucidchart [18].

2.2 Large Language models (LLMs)

LLMs are models trained on vast amounts of textual data that excel in NLP tasks [19]. They started to gain popularity after the release of OpenAI's ChatGPT model in November 2022, which is built using the transformer architecture. It exhibits capabilities such as creative writing, translation, instruction following, summarizing.

While the LLMs can be powerful tools, they are only as good as the data they are trained on. If the training data is bias the outputs will be biases [20]. Therefore, overreliance on LLMs can lead to skill degradation and incorrect assessment [21]. It is important to consider and assess the risk of using LLMs for specific use cases and check their outputs for errors.

2.2.1 Architecture of LLMs

The general architecture consists of multiple layers, including feedforward layers, embedding layers, and attention layers. The Transformer architecture, introduced by Vaswani et al. in 2017 [22], revolutionized NLP by enabling efficient parallel processing of tokens. This capability is achieved its through its multi-head self-attention mechanism, which allows for parallelized model training on large datasets. Their architecture is composed of the following key components:

1. **Input Embeddings**: This layer divides the input text into tokens that are embedded into a continuous vector representation that works as a lookup table.
2. **Positional Encoding**: Encoding does not naturally encode the word order; therefore, positional information is added to each word's vector representation so that it keeps the order of the words as in the text input.

3. **Encoder**: Analyzes the input text and creates several hidden states that preserve the context and meaning of the text data. It processes the input sequence into a representation of all the information the model has learned. This layer consists of two key components:

    - **Multi-head attention**: Self-attention mechanisms are employed, allowing the model to capture different types of relationships and attend to various parts of the input sequence simultaneously.
    - **Fully connected layer**: Processes the attention output further to improve representation. It also includes a normalization component that is applied after each sub-component or layer, helping stabilize the learning process and improve the model's generalization.

4. **Decoder Layer**: The output from the encoder is fed into the decoder, which generates text sequences using two multi-headed attention layers, a feed-forward layer, and a linear layer that acts as a classifier. A SoftMax layer then computes the word probabilities. The decoder runs N times, using the encoder's output and the sentence decoded so far, to ensure coherence in the generated text.

2.3 Techniques used to improve output of LLMs

The quality of a prompt plays a crucial role in determining the output of an LLM. A prompt is a textual input that provides a context to the model, guiding it to generate a response based on pre-trained knowledge. [23]. For optimal performance, prompts must be well-structured, accurate, coherent and contextually appropriate. To enhance the effectiveness of LLM outputs, various techniques have been developed, including prompt engineering, which involves crafting optimized inputs for better responses, and Retrieval-Augmented Generation (RAG), which enhances responses by integrating external knowledge sources.

2.3.1 Prompting engineering

It's very important to find the right prompt, as LLMs are sensitive to prompt design [24]. Finding the right prompt normally requires extensive manual effort. Prompting engineering goal is to find a prompt that achieves the best performance on a given dataset when using a given LLM as the task model.These are the most used techniques:

- **Zero-shot prompting**: involves crafting a single clear and concise instruction without providing examples of the task at hand, taking advantage of the pre-trained knowledge of LLMs. Since it depends on the pre-trained knowledge, it can struggle with complex tasks requiring nuanced understanding. This method is better suited for straightforward tasks and is simple and efficient. It eliminates the need for task-specific fine-tuning, dramatically expanding the range of tasks a model can perform.

- **Few-shot prompting**: introduces a few examples of the task within the prompt to provide better context and clarify expectations. It typically follows a structure of input-output pairs, consisting of an input and the corresponding output, as well as the prompt describing the task the model must perform. This approach leads to better accuracy and increased relevance, as it provides the expected structure of the output. However, there is a risk of overfitting depending on the quality and relevance of the examples.

- **Chain-of-thought prompting**: guides the model to work through intermediate reasoning steps, making it more adept at solving complex tasks such as math problems, commonsense reasoning, and symbolic manipulation. The prompts typically include examples where the problem is decomposed into manageable steps, breaking down complex questions and demonstrating reasoning steps.

2.3.2 Retrieval augmented generation

RAG is a hybrid technique that combines information retrieval with language model generation, to improve the accuracy and contextual relevance of outputs, addressing common issues like hallucination problem [25]. Hallucination occurs when a language model generates inaccurate or fabricated information, particularly when handling queries beyond their training data or requiring current information[26]. RAG mitigates this by integrating external knowledge bases, thereby enhancing the performance of LLMs on knowledge-intensive task [27]. RAG integrates two key components [28] as demonstrate in Figure 2.3.

- **Retrieval mechanism**: This component is responsible for retrieving relevant documents or information from an external knowledge source. Several mechanisms are commonly used, ranging from more traditional methods such as BM25 [29] to more sophisticated techniques like Dense Passage Retrieval (DPR) [30].

- **Generation module**: The generation module processes the retrieved information and produces human-like text by combining the contextual input with the retrieved knowledge.

RAG operates with two distinct types of memory parametric and non-parametric [32]:

- **Parametric memory**: Refers to the internal knowledge already present in the model's parameters during training, represented in the weight embeddings.

- **Non-parametric memory**: Refers to the external knowledge source that the model retrieves during inference. These external sources can include databases, web pages (e.g., Wikipedia), or structured knowledge graphs, enabling the model to answer queries with current or domain-specific information.

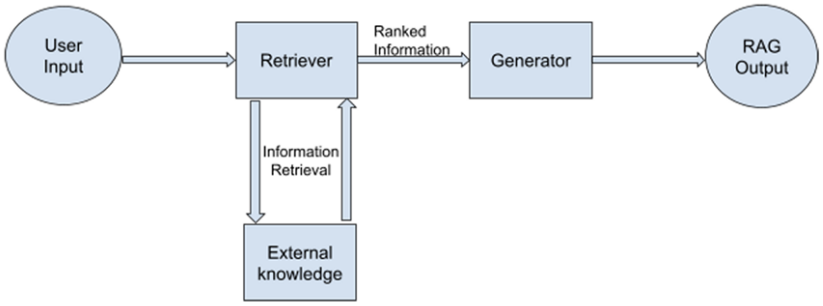


Figure 2.3: Basic architecture of a RAG system [31]

This distinction allows to combine static with dynamic information, significantly enhancing flexibility and accuracy.

RAG models can be classified as two main types, based on how they handle retrieved documents during the generation process [33]:

- **RAG-Sequence Model**: Only the retrieved document is used throughout the generation of an entire sentence or response.
- **RAG-Token Model**: This model allows retrieving different documents for each target token. It selects a distinct latent document for each generated token, retrieves the top k documents, and produces the output sequence probability for each document.

There are three main categories of RAG approaches [34]:

- **Naïve RAG**: The first paradigm of RAG being adopted. It follows a process that includes indexing, retrieving, augmenting, and generating responses (see Figure 2.4).
- **Advanced RAG**: In addition to the steps involved in Naïve RAG, pre-retrieval and post-retrieval processes are included to improve the quality of responses (see Figure 2.5).
  - **Pre-retrieval process**: The main goal is to improve the quality of the content being indexed.
  - **Post-retrieval process**: The main goal is to enhance the way the chunks retrieved are merged with the user query to form the prompt. Techniques such as re-ranking and prompt compression are used to achieve this.
- **Modular RAG**: Divides the retrieval and generation components into independent, interchangeable modules. This allows each module to be independently developed, optimized, or swapped out, enabling experimentation with various retrieval techniques and generation models.

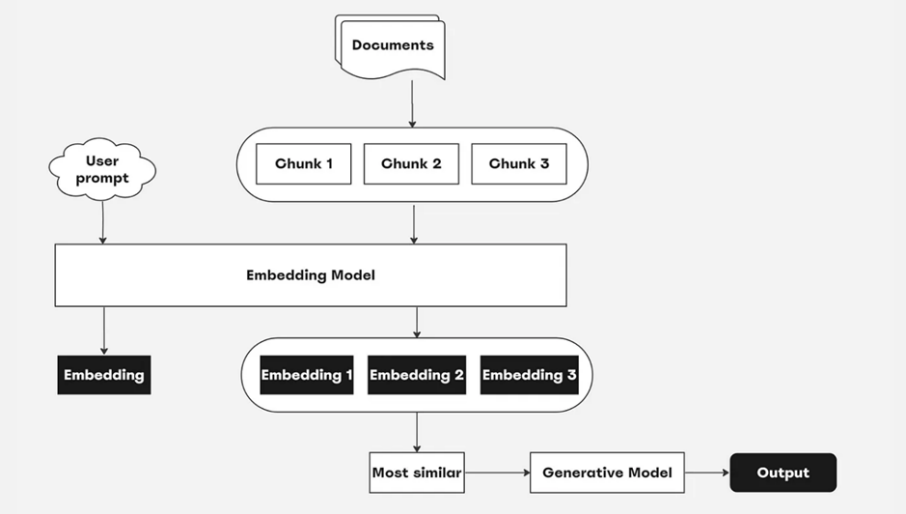


Figure 2.4: Naive RAG [35]

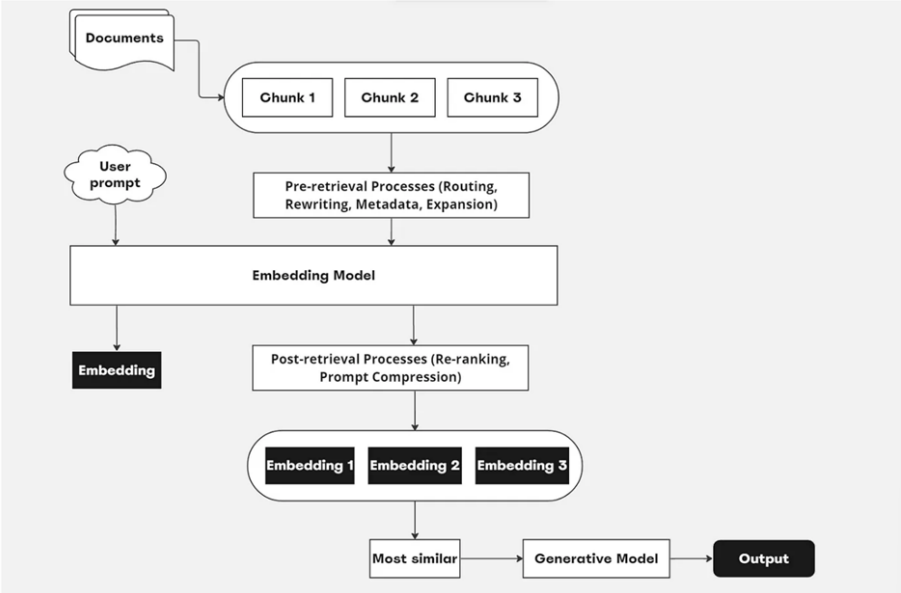


Figure 2.5: Advanced RAG [35]

2.4 Text-based diagram languages

Text-based diagram languages, such as PlantUML and Mermaid, allow to transform graphical representations into structured text, which makes them particularly suitable for LLMs workflows since they work as text-to-text models. These languages use a simple and intuitive syntax to describe various types of diagrams, including UML diagrams, flowcharts, sequence diagrams, and more.

2.4.1 PlantUML

PlantUML focuses on UML and related notations, offering a rich domain-specific language (DSL) for sequence, class, use case, component, deployment, and other diagrams types.

It also supports non-UML diagrams like Gantt diagram and MindMap diagram. In this thesis we will be focusing on class diagrams, therefore we will only give of how to represent class diagrams using PlantUML. A PlantUML representation starts with the opening key word `@startuml` and ends with `@enduml`. Inside this block, we define the classes, their attributes, methods and relationships. A class is declared using `class` keyword followed by the class name. If we want to create a interface or a enum or a abstract class we can use the keywords `interface`, `enum` and `abstract class` respectively. Relationships are representing using different symbols, such as - for associations, <|- for generalization, and *- for composition, and used between the class names. We can label this relationships by adding after the relationships the symbol `:` and the label name. The multiplicity is represented by adding the multiplicity of the class after the class name within quotation marks. The methods and attributes are represented within the class block and we can define the multiplicity using the symbols + for public, - for private and # for protected. In the Listing 2.1 we can see an example of PlantUML syntax representing a library borrowing system and in the Figure 2.6 we can see the generated class diagram from the PlantUML syntax.

```
@startuml
' --- Diagram 7: Library Borrowing System ---
interface User {
  -first_name: String
  -last_name: String
  -borrow_his: List<Borrow>
  +borrow()
  +return()
}

class Adult {
  -id:int
}
class UnderAged {
  -id_student:int
}

class Borrow {
  -date: Date
  -book: Book
}
class Book {
  -author:String
  -title:String
}
class Library {
  -Books: List<Book>
}

User <|.. Adult
User <|.. UnderAged
User "1" -- "1..4" Book : borrow
User "1" o-- "*" Borrow
Borrow --> Book
Library o-- Book
@enduml
```

Listing 2.1: PlantUML syntax example

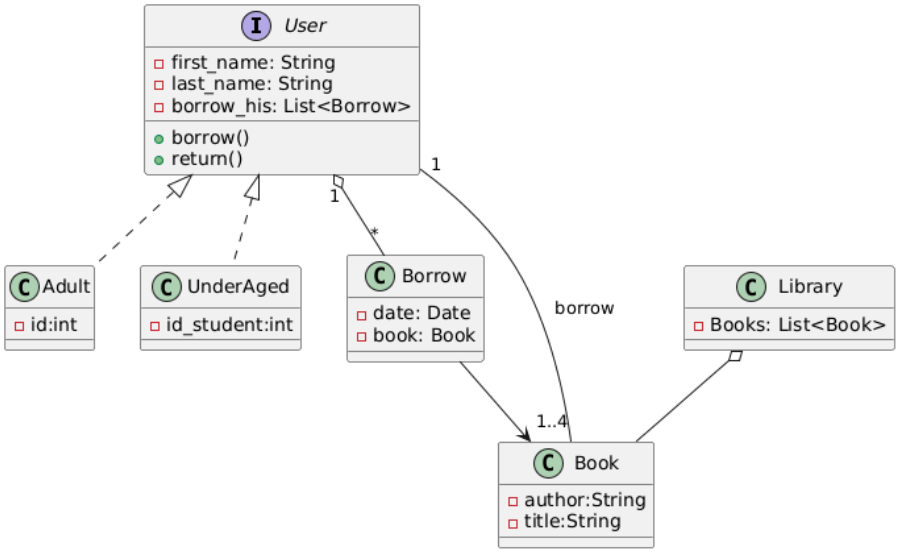


Figure 2.6: Class diagram generated from the PlantUML syntax in Listing 2.1

2.5 Systematic mapping studies

Systematic mapping studies, also referred as scoping studies, are a structured and unbiased methodology for identifying, categorizing, and analyzing the existing literature on a specific research topic [36]. The main goal is to give a wide overview of a research area, to understand the extent of existing studies and identify gaps in knowledge. This methodology consists of systematically searching the literature to determine which topics have been covered, where has been published and in what form [37]. The systematic mapping studies are typically divided into three main stages: planning, conducting and discussion of results. Each stage has a defined process to ensure the reliability and validity of the study. We are going to implement a systematic mapping study to identify the current state of the art in the use of LLMs for requirements engineering, this way understanding better the gaps and possible improvements in the current state of the art.

2.5.1 Planning stage

The first part of the systematic mapping study is to define the research questions that will be driving the research. These questions are crucial, as they are going to guide the search process, help identify relevant studies and establish the focus of data extraction and analysis. Effective research questions, give an overview of the current practices regarding the specific topic and identifying current gaps in knowledge [38].

Additionally, we must define the search strategy to guarantee a comprehensive coverage. This involves composing a search string with a set of keywords based on the main terms of the research questions and driven by their objectives. A properly designed search string assures a search process that is systematic, replicable and unbiased.

Inclusively, we define the inclusion and exclusion criteria with the focus on the research

questions to ensure an appropriate and reliable classification. Inclusion criteria focus on determine studies that address the research questions while exclusion criteria filter out unrelated or irrelevant studies. For example:

- **Inclusion criteria:** Studies that respond to the main research question.
- **Exclusion criteria:** Studies that are in other languages than English.

2.5.2 Conducting stage

In this stage, implement the defined search strategy, by applying the search string in digital libraries and conducting the screening process. Articles that correspond to the exclusion criteria are excluded, similarly, those that provide relevant information, corresponding to the inclusion criteria, are included.

Studies can be considered unrelated to our research just by reading their title and abstract, therefore, they can be excluded without further reading. However, if the abstract isn't clear if corresponds to research questions, the conclusions or any part of the full paper should be read to clarify how it should be classified.

After the initial screening, data extraction is performed. Relevant information from the selected studies is collected, such as methodology implemented, conclusions, publication details, and relationship to the research questions.

Data extraction must be conducted systematically to guarantee the reliability and reproducibility of the results.

2.5.3 Discussion of results

Discussion phase involves analyzing the results of the conducting stage and the responses to the research questions, to gain a deeper understanding of state literature, identify the gaps, trends and underexplored areas on the topic. This phase includes three main tasks, analyzing trends, identifying gaps and documenting results.

For example, a systematic study in natural language processing for requirements engineering might analyze the natural language processing tools being developed in the requirement engineering field, such as OICSI [39].

2.6 Summary

This chapter gives an overview of the main concepts to automating the creation of design models from requirements. It starts by introducing the importance of software modeling and the use of UML class diagrams to represent the structure of a system. Software modeling is explain with a focus on UML class diagrams, explaining their components, relationships, benefits for developers and stakeholders, and describing the design process, highlighting key steps such as requirement analysis, identifying relationships, defining

multiplicity, adding attributes/methods, and refining the model. The chapter then explores LLMs, describing their Transformer-based architecture and key components like multi head self-attention, embeddings and decoders.While LLMs have advanced NLP applications, they face challenges such as context limitations, hallucinations, and dependency on training data. To improve their performance in requirement modeling, two techniques are examined: prompt engineering (zero-shot, few-shot, and chain-of-thought prompting) and RAG, which integrates external knowledge to enhance accuracy. Finally, the chapter discusses systematic mapping studies, a structured method for analyzing existing research. It describes the planning, conducting, and discussion stages of the study conducted in this work, which aims to identify gaps in LLM-based requirement modeling. These insights serve as the foundation for the research, helping define strategies to enhance LLM-driven software modeling automation.

3

State of the Art

This dissertation explores the automation of requirements modelling, with the systematic mapping study being useful to identify the state of art and understand possible improvement to implement. The systematic mapping study provides an overview of existing knowledge on the topic while uncovering gaps that require further exploration [37]. It starts with the specification of the research questions and the implementation of the research methodology. Then, a search strategy is defined to detect relevant literature, that in turn will be submitted to an inclusion and exclusion criteria and a quality assessment to select the final primary studies. Lastly, data from the selected studies is extracted and analysed to assess the existing evidence regarding the topics of the previously defined research questions. The results offer insight into the advancements and limitations in the field, giving a better understanding of possible improvements in automations of requirements modelling. This chapter outlines the steps taken to conduct the systematic mapping study and elaborates on the key findings that inform this dissertation.

3.1 Research Methodology

The overall approach of the research methodology was to analyze current approaches used to automate requirements modeling and design modeling, with a particular focus on methods utilizing LLMs. The goal is to explore the capabilities and challenges of leveraging LLMs in transforming requirements into design models, as well as their role in automation within the software development modeling process.

To achieve this, we conducted a systematic mapping study methodology compose by three key stages: planning, conducting, and reporting. The planning stage involved establishing research questions, defining search and study selection strategies, performing a quality assessment of studies, and developing a data collection and extraction strategy. In the conducting stage, we apply the plan prepared and analyse the preliminary results obtained from the execution of the search strategy. Finally, the reporting stage included a thorough analysis and discussion of the findings.

3.1.1 Planning Stage

The first phase of the systematic mapping study, comprises five main activities defining research questions, formulating a search strategy, establishing a study selection approach, developing a quality assessment strategy, and determining data collection and extraction methods.

Research Questions The objective of this study is to provide a comprehensive overview of the existing approaches for automating design modeling. Consequently, the research questions were designed to address key aspects of automating design modeling using LLMs and GAI. These questions aim to offer a structured understanding of the state-of-the-art in this domain while identifying gaps and opportunities for future improvements. In the Table 3.1, the research questions and the motivation behind are presented.

Research Questions	**Motivation**
RQ1. How have LLMs been used to derive design models from requirements?	To investigate how LLMs are applied to transform textual requirements into design models, providing insight into practical applications.
RQ2. What are the strengths and limitations of using LLMs in this context? (deriving design models from requirements)	To gain an overview of the potential benefits and challenges of using LLMs.
RQ3. What GAI tools have been applied for producing class diagrams and object-oriented artifacts in software development?	To identify which GAI models are most commonly used.
RQ4. What are the key approaches in automating functional requirements into design models using AI?	To discover methodologies that leverage AI to automate the process of requirement modeling.

Table 3.1: Research Questions

Search Strategy The search strategy involved identifying primary studies using an automatic search method. The search string was constructed with three key elements,examples of LLMs widely used in the industry, terms conveying the concept of requirement modeling, and synonyms for methodologies.The search string is presented in Table 3.2.

The search string was run as a query on IEEE Xplore [40] and ACM Digital Library [41]. These libraries were selected for their comprehensive collections of peer-reviewed publications in software engineering and their user-friendly interfaces that allow bulk downloading of search results in CSV format. Besides the search question we implemented a filter to only show papers from 2019 and foward because we believe that the most

Concept	Keywords
Large Language Models	("Large Language Models" OR "LLMs" OR "ChatGPT" OR "LLAMA" OR "Gemini")
Requirement Modeling	("design model*" OR "requirement model*" OR "class diagram" OR "requirement specification" OR "architecture model" OR "component diagram" OR "sequence diagram")
Related terms for methodologies	("approach" OR "method" OR "technique" OR "tool" OR "process")

Table 3.2: Search keywords for requirement modeling

relevant papers are the most recent ones and do to the limit time-line we had to conduct this systematic mapping study.

Study Selection Strategy Studies identified through automatic searches were evaluated using inclusion and exclusion criteria to ensure relevance to the research questions. The evaluation was conducted in two iterations. The inclusion and exclusion criteria used in the study selection process are presented in Table 3.3. Given the prevelance of English in the scientific community, we decided to only select papers written in English. We also excluded duplicate papers, informal literature, and secondary studies to ensure the quality and relevance of the selected studies. The study selection process was conducted in two iterations. In the first iteration, the title and abstract of each study were reviewed, selecting those that met all inclusion criteria while discarding those that matched any exclusion criterion. In the second iteration, the full text of the pre-selected studies was analyzed, ensuring that only those fully aligning with the inclusion criteria were retained as final primary studies.

ID	Inclusion Criteria
IC1	Papers that answer at least one the research questions.
IC2	Peer-reviewed papers from journals, conferences, and workshops.
	Exclusion Criteria
EC1	Papers not written in English.
EC2	Duplicate papers, such as short and long versions of the same study.
EC3	Informal literature (e.g., slide shows, conference reviews, informal reports), secondary studies (e.g., reviews, surveys), and studies without practical applications.

Table 3.3: Inclusion and exclusion criteria

Data Collection and Extraction To simultaneously collect all the information needed to answer the research questions and manage the data that was extracted, we created a spreadsheet to record the content of each selected study. This way, we can ensure that all selected papers are subjected to the same extraction criteria.The spreadsheet can be accessed through the following link.

3.1.2 Conducting stage

The process to identify and collect the studies that provide answers to our research questions was conducted by following the steps of the Planning Stage 3.1.1. As mentioned previously, initially, was performed the automatic search by using a predefined search string on the IEEE Xplorer and ACM Digital Library, yielding a total of 419 papers from ACM and 591 from IEEE. Next, we read and analyzed the titles and abstracts of each potential paper to apply the first iteration of inclusion and exclusion criteria, narrowing the selection down to 32 papers. These 32 papers where then read in full during second iteration, leading to the final 15 papers. Lastly, we validated and extracted the data that answered the research questions from all the selected papers. The figure 3.1 summarizes the process and the table of the papers selected is presented in appendix A.

3.1.3 Discussion of the results

In the following subsections, we answer to each research questions, based on the analysis of the data extracted from the selected studies.

RQ1: How have LLMs been used to derive design models from requirements?

Through the analysed papers, several methods were identified in which LLMs have been employed to derive design models from requirements. Direct code generation is one of the most used methods, studies such De Bari et al. [42], Jahan et al. [3], and Ferrari et al. [6] used ChatGPT to generate PlantUML [17] code directly from requirements and user stories. Babaalla et al. [4] employed multiple models, including GPT-2, BERT and LLaMA-2, to transform textual tokens into vector representations. Basically encoding entities such as classes attribute and relationships. This vectors are then inputted into layers trained on bidirectional Long-short-term memory (LTSM) recurrent neural networks and tree structures. Eisenreich et al. [43] proposed utilization of LLMs to generate domain models, scenarios and architecture candidates, incorporating iterative human interactions to refine the outputs and evaluate architecture options. Requirements elements extraction is also used by Ruan et al. [7], Omar et al. [8] and Li et al [44]. The proposed platform " Wisdom Requirement Communication" presented by Wang el al. [45] also does requirement elements extraction. Also, using techniques such as tokenization and part-of-speech tagging to pre process the requirements, it uses GLM 4-LLM to conduct advanced requirements analysis, identifying actors, use cases, and processes which are then converted into Mermaid format for visualization. Bernadi et al.

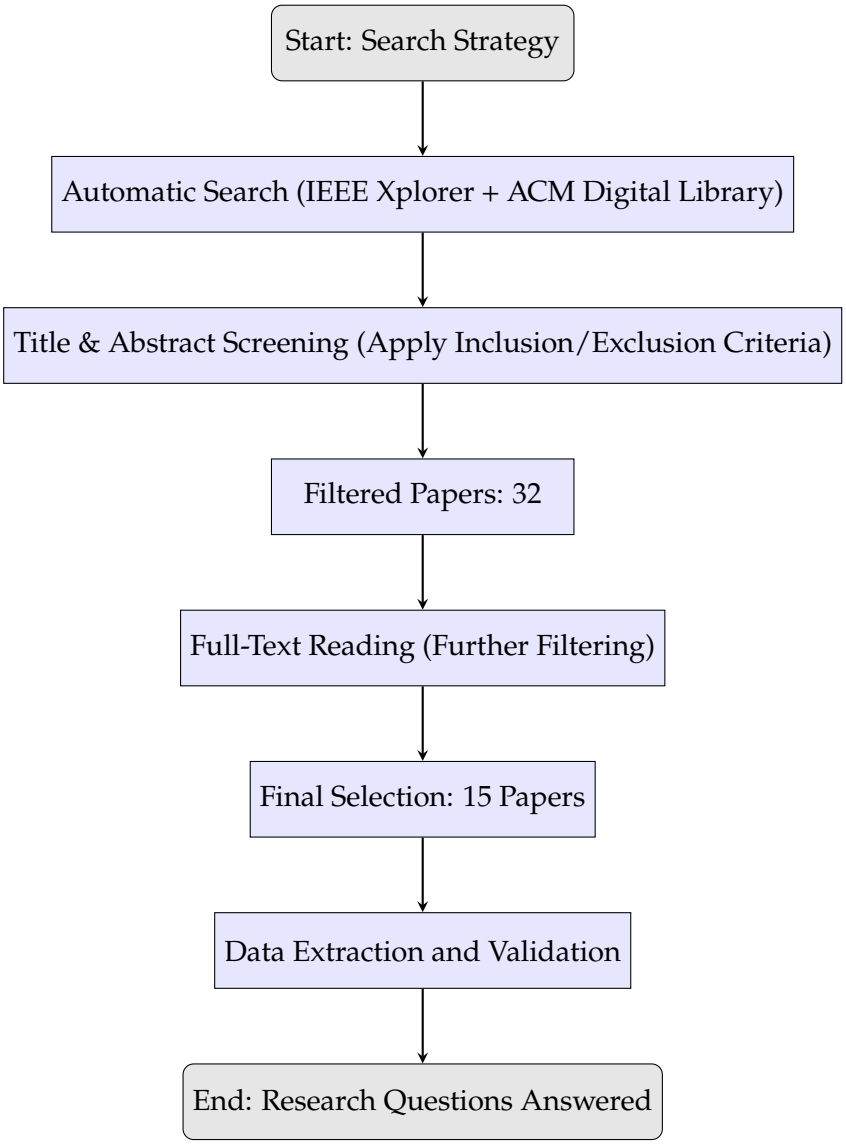


Figure 3.1: Process of Identifying and Collecting Studies

[46] combined LLaMA models with a Retrieval Augmented Generation (RAG) system to analyze and enhance UML diagrams, leveraging domain-knowledge to improve feedback. Chaaben et al. [47] applied LLMs to complete domain models by transforming incomplete models into textual representation using semantic mapping. These representations were use to generate suggested modelling elements missing, which were refined by applying a frequency-based ranking.

RQ2: What are the strengths and limitations of using LLMs in this context?

The strengths of using LLMs in this context identify are automation of design process, by allowing to generate UML diagrams, design models, architectural components and evaluation scenarios from textual requirements it reduces manual effort [3, 42, 43, 48]. Using prompt-based techniques enhanced efficiency of creation of design artifacts [44, 47].

Method/Approach	Studies	Techniques and Models employed	Output/Design Model Derived
Direct Code Generation	[3, 6, 42]	ChatGPT is used to directly generate PlantUML code from requirements and user stories.	PlantUML code representing design models.
Vector-Based Representation & LSTM Analysis	[4]	GPT-2, BERT, and LLaMA-2 transform textual tokens into vector representations; these vectors are then fed into bidirectional LSTM networks and tree structures.	Encoded entities (classes, attributes, relationships) that inform design models.
Iterative Domain Model Generation	[43]	LLMs generate initial domain models, scenarios, and architecture candidates, refined through iterative human interactions.	Refined domain models, scenarios, and architecture candidates.
Requirements Elements Extraction	[7, 8, 44]	LLM-based extraction techniques to identify key requirement elements.	Identification of actors, use cases, and processes from requirements.
Advanced Requirements Analysis	[45]	Utilizes tokenization, part-of-speech tagging, and GLM 4-LLM for advanced requirements analysis; outputs are converted into Mermaid format for visualization.	Visual representations (via Mermaid) of actors, use cases, and processes.
UML Diagram Enhancement via RAG	[46]	Combines LLaMA models with a Retrieval Augmented Generation (RAG) system to integrate domain knowledge and enhance UML diagrams.	Enhanced UML diagrams with improved feedback based on domain knowledge.
Domain Model Completion	[47]	Transforms incomplete domain models into textual representations using semantic mapping; suggests missing modeling elements refined by frequency-based ranking.	Completed domain models with additional suggested modeling elements.

Table 3.4: RQ1:Methods for Deriving Design Models from Requirements Using LLMs

General-purpose LLMs like GPT-4 Chat-3.5 and LLaMA can be applied across various domains, making them versatile for software modelling task [43, 45] LLMs can produce code in formats like PlantUML and Mermaid can produce code in formats like PlantUML and Mermaid which give visualization support [8, 43]. LLMs gives flexibility in model generation by having the ability to extract and represent complex entities, attributes and relationships from natural language [4, 7].

On the other hand, LLMs show limitation such as semantic and domain knowledge gaps [6, 8, 42, 45]. Also, LLMs show issues with relationships, often incorrectly identifying and modelling relationships between entities [4, 9] or omitting key elements or misrepresenting it. [3, 47]. LLMs tend to incompletely handle requirements, due to complex or

incomplete input conditions [6, 43] or ambiguities and inconsistencies in textual requirements [7, 47]. The outputs are highly dependent on the quality of prompts are require expert guidance make harder the possibility of fully automated processes [9, 46]. Another limitation is the variability in the performance, as the results depend on factors such as input formats, prompting techniques, and the specific LLM versions being used.This leads to inconsistent performance in generations of modelling features like abstract classes or precise architectural patterns [4, 47, 48].Additionally, the performance is strongly influenced by the training of each model [49]. The effectiveness of LLMs depends heavily on the availability and quality of external knowledge bases and example is the approach Bernardi et al. that improve the quality of the model generated by using RAG techniques [46].

Strengths	Key References
Automation of design process	Reduces manual effort by generating UML diagrams, architectural components, and evaluation scenarios from textual requirements [3, 42, 43, 48].
Improved efficiency via prompt-based techniques	Enhances speed of creating design artifacts [44, 47].
General-purpose applicability	LLMs (e.g., GPT-4, ChatGPT-3.5, LLaMA) are versatile across multiple domains [43, 45].
Visualization support	LLMs can produce code in PlantUML or Mermaid formats for diagram generation [8, 43].
Flexibility in model generation	Can extract and represent complex entities, attributes, and relationships from text [4, 7].

Table 3.5: RQ2: Strengths of using LLMs in software modeling

Limitations	Key References
Semantic and domain knowledge gaps	May misunderstand complex domain-specific information [6, 8, 42, 45].
Incorrect or omitted relationships	Often misidentifies or omits key relationships among entities [3, 4, 9, 47].
Incomplete handling of requirements	Struggles with complex, ambiguous, or inconsistent textual requirements [6, 7, 43, 47].
Reliance on prompt quality and expert guidance	Outputs depend heavily on carefully engineered prompts, limiting full automation [9, 46].
Performance variability	Depends on input formats, prompting techniques, LLM versions, and training data, leading to inconsistent results [4, 47–49].
Dependency on external knowledge bases	Quality can improve through Retrieval-Augmented Generation (RAG) or similar methods (e.g., Bernardi et al.) [46].

Table 3.6: RQ2: Limitations of using LLMs in software modeling

RQ3: What GAI models have been applied for producing class diagrams in the

context of software development?

The most use GAI model was the one developed by OpenAI, ChatGPT, different versions of the model were used in different applications [50]. The second most popular was LLaMA developed by Meta [51]. Also, in the Wang framework [45] it was used the GLM-4 model, developed by THUDM (Tsinghua University Distributed Machine Learning group) and supported by Zhipu AI [52].

GAI model	Papers
ChatGPT models	[2–4, 6–9, 42–44, 47, 48]
LLaMA models	[4, 43, 46]
GLM-4 model	[45]

Table 3.7: R3: Generative AI models used in the papers analysed

RQ4: What are the key approaches in the automation of functional and non-functional requirements into design models using AI?
On of the key approach was using LLMs for automate the design models. The use of prompting engineering are fundamental in order to enhance the extraction and synthesis of design. Chaaben et al. use few-shot prompt learning to overcome the scarcity of specialized datasets required for traditional deep-learning approaches [47]. Ruan et al. [7] proposed framework uses zero-shot learning to extract the elements of requirements descriptions. Or the study of Chen et al. [7] that compares the performance of different prompting engineering. The prompt employed included a task description a detailed domain specification, a format description and, depending on the technique used, reference examples with reasoning steps to guide the model's response. Another is a rule-based approach like the one presented in the study by Jahan et al. that uses preprocessing methods to deal with compound sentences and grouping consecutive nouns, then applying NLP heuristic that extract the elements from the requirements. He then compared a rule-based approach with a LLM approach [3]. A machine learning hybrid approach such as the one used by Babaalla in his study where it was combined bidirectional LSTMs, tree structures and LLMs. This multi-layered architecture jointly extracts UML fragments (e.g., classes, attributes, methods) and relationships, thereby mitigating error propagation often seen in staged approaches [4]. Another Hybrid approach was developed by Eisenreich et al. [43] and Ahmad et al. [48] that implemented iteratively refinement through human feedback in their approaches. Another approach was the use of Retrieval- Augmented generation. Bernardi et al. used Retrieval-Augmented Generation (RAG) to improve the domain knowledge of the LLM in order to improve the feedback about the models [46].

Approach	Studies	Techniques Employed	Key Aspects/Outcome
LLM-based Prompting	[2, 7, 47]	Few-shot and zero-shot learning; prompts including a task description, detailed domain specification, format guidelines, and (when applicable) reference examples with reasoning steps.	Enhances the extraction and synthesis of design models from requirements.
Rule-based Approach	[3]	Preprocessing techniques (handling compound sentences, grouping consecutive nouns) combined with NLP heuristics.	Extracts requirement elements; used as a baseline for comparing LLM approaches.
Hybrid ML Approach	[4]	Integration of bidirectional LSTMs, tree structures, and LLMs to jointly extract UML fragments (classes, attributes, methods) and relationships.	Mitigates error propagation typical of staged approaches.
Hybrid with Iterative Human Feedback	[43, 48]	Iterative refinement of LLM outputs through human feedback.	Improves the accuracy and relevance of the generated design models.
Retrieval-Augmented Generation (RAG)	[46]	Combines LLMs with a retrieval system to enhance domain knowledge integration.	Enhances model feedback and improves the quality of generated designs.

Table 3.8: R4: Key Approaches in the Automation of Functional and Non-Functional Requirements into Design Models Using AI

3.2 Synthesis of the results

The results of the systematic aimed to understand the current state of research on the use of LLMs and GAI for automating requirement modelling. We have identified that there are hybrid approach that take leverage of machine learning models [4], others introduce a iterative human-refinement in order to improve the result [43, 48], or rule-based methods to improve extraction of design elements [3], or a implementation of a RAG system to improve the domain knowledge of the LLM [46]. A common factor was the importance of good prompts to achieve good results. Therefore, many of the studies employed prompt engineering techniques such as zero-shot learning [7], few-shot learning [47] and CoT [2]. The analysis showed that the most frequently used LLM was ChatGPT, developed by OpenAI. Its widespread application shows its strong versatility and robust performance across various task. Following closely, was LLaMA , owned by Meta. Its frequently used in task requiring domain-specific understanding. It's noticing that the LLMs can improve the manual effort in requirement modelling. However, they still face limitations. These includes difficulties identifying relationships between entities, variability in output quality, struggle with handling domain-specific knowledge, and poor performs when dealing with ambiguous or incomplete requirements.

3.3 Threats to validity

Internal Validity In systematic mapping studies there is always the risk of not including all relevant studies. While defining the search string, we made an effort to include synonyms and alternative terms related to the research questions. Although this efforts, the results may miss some relevant studies that do not use the specified keywords. To mitigate this risk, we text several search strings based on the number of relevant results they retrieved and chose the most optimal.
Additionally, there is the potential risk of excluding relevant studies during the selection. As the study selection and data extraction were conducted by a single person, and the volume of studies was substantial, the inclusion criteria and exclusion criteria were defined to avoid, at least to some extent, introducing selection bias in the process.

External Validity To ensure we had the maximum of relevant research work regarding our goal, we obtained the studies through the digital libraires, IEEE Xplore and ACM Digital Library, that are well established digital libraries in the software engineering field and ensured a large set of peer-to-peer reviewed papers.However, a potential limitation arises from the exclusive reliancce on these two digital libraries, as other relevant studies may be present in different sources such as Springer and ArXiv were not considered. This may have led to the omission of relevant studies, introducing a potential bias in the selecting process. Additionally, we did not consider Non-English or papers published before 2019. While this constraints were implemented to maintain focus and ensure high-quality sources, they introduce a risk of missing valuable contributions outside our selected databases and timeframe.

Conclusion Validity A challenge to validity was the fact we do not have the knowledge of how the search engines of the used digital libraries work. To address this, we documented the exact search string used and we extracted a csv file with the results of the search to ensure consistency across searches and to facilitate systematic data organization. However, as the study selection and data extraction were conducted by one person, there remains a potential for subjective bias. Although these challenges, the structured methodology and tools used contribute to the reliability of the conclusions drawn.

3.4 Related Work

NLP and LLMs have show potential to change the software engineering field, particularly in requirements engineering, architecture design, conceptual modelling, taxonomy construction, code generation and test case generation. The systematic study on NLP for requirements Engineering (NLP4RE) [53] published in 2021 identifies the analysis phase as the most researched followed by management, elicitation, and modeling, while validation verification, testing, and design receive the least attention. The study also identifies that

requirements specification documents are the most common input, and that tools developed are mainly focused on modelling, detection and extraction, although only 13.08% of the tools are accessible online. An the most widely used NLP technologies include POS tagging, tokenization, parsing with tools such as StanfordCoreNLP, GATE, and NLTK. A significant challenge discovered on the systematic study is the absence of RE-specific resources, which prevent the development of effective domain-specific solutions. The authors belove that transfer learning techniques can help addressing this gap.

LLMs have been employed for different tasks related to software engineering. One of them is generate architectural designs [54], in this paper the authors used three different approaches (zero-shot, few-shot prompting, and fine-tuning), and different models to evaluate the performance of the LLMs. The results showed that GPT-4 excels in generating structured and relevant ADDs. However, smaller models can achieve comparable results. After fine-tuning, Flan-T5-Base with fewer parameters achieves comparable performance to GPT-3.5 in a few-shot approach, suggesting its utility for organizations prioritizing cost efficiency, privacy, and minimal hosting infrastructure. Although, LLMs are not entirely reliable for ADR generation but can effectively assist architects in documenting ADDs. Despite advancements, even the best fine-tuned models cannot match GPT-4 in adhering to the ADR markdown format. The findings suggest that smaller models like text-davinci-003, when used with a few-shot approach, can offer a cost-effective alternative to larger models like GPT-4, although their generalizability remains uncertain. This underscores the trade-offs between cost, infrastructure, and performance when using LLMs for ADD generation.

Similar, Chen's at el. [55] compares the use of prompting and fine-tuning to enhance the performance of Large Language Moldes for taxonomy construction. The users used two techniques to fine-tune the model, layer-wise fine-tuning, which updates parameters in the last few layers, and LoRA, which uses low-rank decomposition to update all parameters. In contrast for prompting, to deal with variation of outcomes when generating taxonomies with prompting, they use majority voting to aggregate outcomes from multiple runs. The results of the comparative study indicate that the prompting method outperformed fine-tuning when compared using F1 score. The gap between the two approaches increases when the training dataset is smaller because is more challenging to fine-tune. Although, neither approach surpassed the current state-of-art performance achieved by BERT-based models on WordNet, suggesting that model architecture significantly influences task performance. The study emphasizes the importance of selecting appropriate methods based on dataset size and resource constraints while acknowledging the need for further advancements to address existing limitations in taxonomy generation using LLMs.

Another task were LLMs can be used is in establishing traceability between natural language requirements and software. Ali at el. approach [56] used LLMs combined with RAGs to compact the ambiguities within requirements engineering. It offers the knowledge to be dynamically incorporated in a scalable, plug-in fashion compared to traditional models in which the knowledge is parametrized. Another advantage is that

by using human-authored texts incorporate in the generation, it simplifies the generation process and may improve out quality. The effectiveness of RAG relies heavily on the quality of its index, which serves as a contextual bridge, helping LLMs disambiguate requirements by providing precise context. The RAG system was implemented in two stages: the indexing stage, which creates and stores code summaries as indexes for fast and relevant matches, and the querying stage, which retrieves relevant documents to provide contextual input for LLMs in predicting classes aligned with a given requirement. This approach ensures that retrieved documents demonstrate high relevance accuracy, which is critical for reliable requirements traceability. By prioritizing precision over recall, the RAG mechanism minimizes irrelevant links between requirements and code, reducing the risks of misleading connections and supporting efficient maintenance and verification processes.

Similarly, Arora's at el. approach [57] also employs a retrieval augmented system to generate a test scenario form natural language requirements. The choice of employing a RAG system is do to LLMs facing challenges with domain-specific queries, often producing inaccurate or incomplete outputs. By incorporating domain-specific context into prompts, improves the accuracy and relevance of generated TSs. The industrial case study presented by Arora's [57] in collaboration with Austrian Post Group IT, revealed that bilingual and domain-specific requirements, such as proprietary internal processes, are particularly challenging for LLMs due to limited public data. As in Dhar's et al. explanatory empirical study [54] it was used the BLUE, ROUGE and METEOR evaluation metrics, providing unique insights into similarity, recall, and alignment with human judgment. A notable indirect advantage found in this study was that when the generated tested scenarios presented major issues, it was due to the original requirements were incomplete and poor-quality. In both studies the approach presented great promises, but which are constrained by the quality of input, preciseness of prompts and effective contextual integration.

A exploratory study conducted on the task of using LLMs [58] for conceptual modelling showed that the user is aware of the lack of contextual memory, inaccuracy and unreliability reliance on precise inputs of LLMs, aligned with those observed in [55]. Addressing these limitations is essential to improving usability and reliability across application domains.

In conclusion, he integration NLP and LLMs into the different areas of software engineering domain shows potential to transform those areas. Despite the advancements made indifferent areas, several common themes and challenges emerge across studies. First, the lack of domain-specific resources and datasets is recurring limitations as observed in systematic mapping NLP4RE [53], taxonomy generation [55], and test case generation [57]. This restricts the ability to create reliable and effective tools for software engineering tasks. Second, there is a trade-off between cost, infrastructure, and performance. Smaller models often are cost-effective solutions but often fall short in terms of generalization and accuracy compared to large models like GPT-4 Additionally, contextual integration plays a critical role in effectiveness of these models. Whether through RAG systems [54, 57], or the structured use of prompts, incorporating domain-specific knowledge significantly

improves the relevance and accuracy of outputs. Nevertheless, these approaches depend on the quality of input data and the precision of prompt engineering. As suggested in by Ali's [58] its fundamental to address limitations such as lack of contextual memory, inaccuracy, and dependence on precise inputs in LLMs, to improve usability, reliability, and integration into broader software engineering practises.

3.5 Summary

This chapter describes the systematic mapping study done during this first part of this thesis. The systematic mapping study aimed to understand how the LLMs have been used for deriving design models from textual requirements. The research methodology applied followed a three key phases: planning, conducting, and reporting. The study identified various techniques used to leverage LLM in design modeling. Direct code generation was one of the most common approaches, where LLMs like ChatGPT and LLaMA were used to generate UML diagrams from textual descriptions. A significant finding was the increasing integration of RAG systems, which improve LLMs' capabilities by integrating domain-specific knowledge. However, the study also highlighted key challenges, including the ambiguity and incompleteness of natural language requirements, the dependency on high-quality prompts, and the limitations of LLMs in capturing complex software design patterns. Despite the promising applications of LLMs, challenges such as limited contextual memory, variability in output quality, and difficulty in handling domain-specific knowledge remain a struggle. These findings underscore the need for improved prompt engineering techniques, more structured data augmentation, and further advancements in contextual awareness and domain specific knowledge integration to enhance the usability and reliability of LLMs in design modeling.

4

Experiments with Generative AI techniques and tools

This chapter reports the preliminary experiments conducted to configure an automated pipeline that generates UML class diagrams from natural-language system descriptions. We compare prompt-engineering strategies and large language models (LLMs), and we evaluate Retrieval-Augmented Generation (RAG) settings to identify the combination that best supports this task. We also test a self-refinement loop in which the model critiques and revises its own output to improve quality.

4.1 Experiments with prompt techniques and different LLM models

To probe how prompt engineering and model choice interact when converting natural-language requirements into UML class diagrams, we conducted a controlled experiment structured according to the setup represented in Figure 4.1. The experiment was centred on a single, medium-sized specification drawn from the dataset of Daniele De Bari et al. [59]. We decided to use this dataset because, besides having the system description and the associated benchmark class diagram, it also has a class diagram that was created by a human. By having these two class diagrams we can to have a deeper evaluation of the generated diagrams. Listing 4.1 shows an system description used; it contains multiple actors, artefacts, and life-cycle constraints—enough complexity to exercise class identification, attribute extraction, and relationship mapping. After the dataset was defined, we designed an experiment to systematically evaluate the impact of different prompt strategies and model choices. The experimental setup is described below.

```
A project manager uses the project management system to manage a project. The project
    manager leads a team to execute the project within the project's start and end dates.
Once a project is created in the project management system, a manager may initiate and
    later terminate the project due to its completion or for some other reason.
As input, a project uses requirements. As output, a project produces a system (or part of
    a system). The requirements and system are work products: things that are created,
    used, updated, and elaborated on throughout a project.
Every work product has a description, is of some percent complete throughout the effort,
    and may be validated.
However, validation is dependent on the type of work product. For example, the
    requirements are validated with users in workshops, and the system is validated by
    being tested against the requirements.
Furthermore, requirements may be published using various types of media, including on an
    intranet or in paper form; and systems may be deployed onto specific platforms.
```

Listing 4.1: System Description

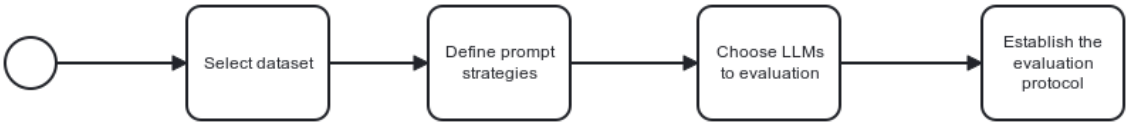


Figure 4.1: Experimental setup for prompt techniques and LLMs tests

Define prompt strategies Each model was queried under three canonical strategies:

1. **Zero-shot** – task instruction only;
2. **Few-shot** – three worked examples followed by the target specification;
3. **Chain-of-thought Few-shot (CoT)** – the few-shot template augmented with an explicit, step-by-step reasoning scaffold.

The prompts used for testing can be found in the Appendix B.

Choose LLMs to Evaluate We considered five commercially hosted models that can be accessed without a local GPU:

1. ChatGPT-4o[1]
2. Claude Sonnet 4[2]
3. DeepSeek XL[3]
4. Gemini 2.5 Flash[4]

[1] OpenAI ChatGPT-4o
[2] Anthropic Claude Sonnet 4
[3] DeepSeek XL
[4] Google DeepMind Gemini 2.5 Flash

5. Grok 1.5[5].

Several open-source models (Mistral 7B, Qwen 7B, Falcon 7B) were piloted but discarded because the results were weak. Some generated PlantUML code with syntax errors that did not compile, while others were not able to generate PlantUML code at all. Besides this, when we used few-shot prompting, some models generated class diagrams not related to the system description but instead to the examples presented in the prompt. We also tried open-source models with more parameters; although they achieved better results, they still performed significantly worse than the commercial models. Therefore, due to weak results and hardware limitations in running powerful open-source models, we ended up discarding them for these tests.

Evaluation protocol The generated diagrams were assessed along three complementary axes:

1. **Structure.** We compared each diagram to the ground-truth model via a confusion matrix over classes, attributes. We defined as true positives all the elements that were presented in the generated diagram and the benchmark diagram. False positives were the elements that were present in the generated diagram but not in the benchmark diagram. False negatives were the elements that were present in the benchmark diagram but not in the generated diagram. From these values, we computed the precision , recall, and F1-score using the formulas bellow:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \qquad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \qquad F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.1}$$

2. **Semantic and pragmatic adequacy.** We adopted G-Eval [60] with Llama-3 as the judge, requesting two rubric items: *semantic fidelity*, which rates the correctness and completeness of domain concepts and relationships and *pragmatic usefulness*, which rates clarity of the diagram, appropriate use of the UML symbols and multiplicities.

3. **Lexical overlap.** ROUGE-L recall [61] was computed between the generated PlantUML and the reference code. ROUGE-L measures the longest common subsequence of tokens between two texts, normalized by the length of the reference; in this context, higher recall indicates that fewer tokens (for example, attributes or class names) have been omitted. Although it is a surface-level metric, it helps reveal omissions that may not be fully captured by the structural counts (e.g., missing attribute declarations).

4.1.1 Results and Discussion

Figures figs. 4.2 to 4.4 and figs. 4.5 to 4.7 show the outcome of the experiment. We analyze the main findings below.

[5] xAI Grok 1.5

Prompt strategy effects. CoT delivers the highest mean G-Eval and the strongest recall, as it can be confirmed in Figure fig. 4.2, confirming that an explicit reasoning trace helps models surface low-salience entities such as *WorkProduct*. However, its ROUGE score lags behind few-shot because the additional prose dilutes lexical overlap with the benchmark UML class diagrams as it can be seen in Figure fig. 4.4. Few-shot, in contrast, attains the best ROUGE and the second-best recall, suggesting that concise demonstrations strike a balance between coverage and verbosity. Zero-shot performs adequately on ChatGPT-4o but drops in both recall and G-Eval for the remaining models.

Model behaviour

- **Claude Sonnet 4** consistently tops the G-Eval table and ranks second in structural recall, making it the most balanced performer.
- **Deepseek** yields the highest recall and F_1, although with slightly lower G-Eval. This is an indicator that it enumerates more elements but produces output with less refinement compared to Claude.
- **ChatGPT-4o** excels in zero-shot but gains little from additional prompting.
- **Gemini 2.5 Flash** shows moderate scores across metrics, hampered by occasional length truncation in CoT mode.
- **Grok 1.5** underperforms structurally and, in CoT mode, returned an image instead of code, highlighting alignment gaps for specialized tasks.

4.2 Experiments with a Retrieval-Augmented Generation (RAG) Tool

To further understand if implementing a RAG system would be beneficial for generation of class diagrams we conducted experiments using the new google tool, NotebookLM[6]. NotebookLM grounds its responses in a user-supplied document collection, thereby enabling context-aware retrieval during generation [62]. However, it does not allow to personalize the underlying components of a RAG architecture, such as the retriever strategy and chunking, limiting the optimization for specific tasks. This way is difficult to understand how the data is being handled, if it is being properly indexed and retrieved.

4.2.1 Experimental Setup

To conduct this experiment we configured a notebook by adding the following documents. Aiming to simulate the reference material available to a professional modeller:

[6]Formerly "Project Tailwind," an AI-assisted research and note-taking environment released by Google Labs.

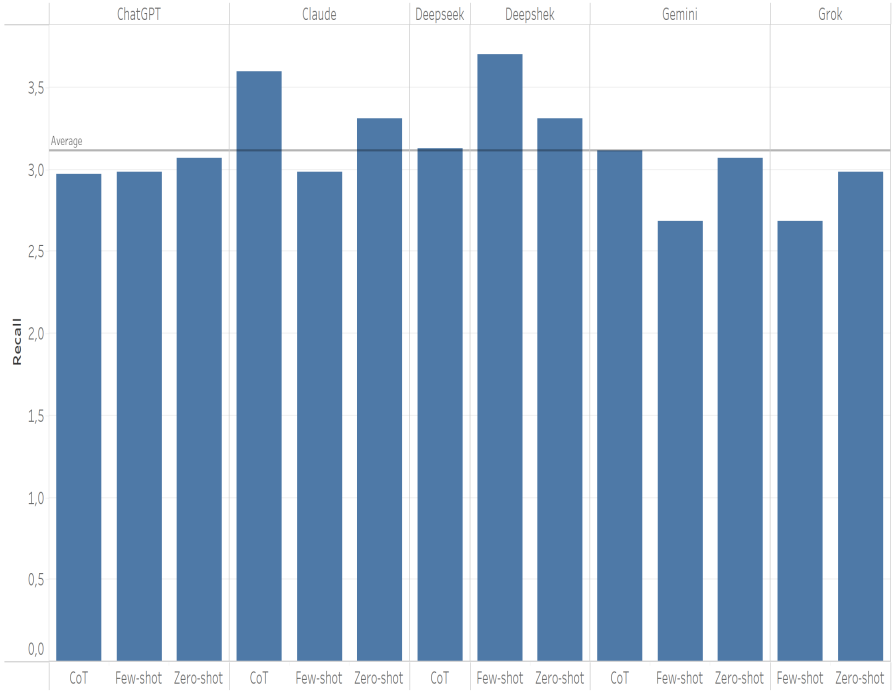


Figure 4.2: Recall of each LLM model per prompt technique.

1. Buku-learning UML 2.0 - A pedagogical guide to UML concepts [63].
2. OMG UML specification - The official UML language specification [64].
3. PlantUML reference guide - Syntax and rendering documentation for PlantUML [65].
4. Examples from the UML modeling dataset - Simulating models created in previous projects [59].

An important limitation found is that the NotebookLM imposes constraints on the number of tokens, limiting the possibilities of using prompting engineering. Therefore, we were only able to use zero-shot prompt supported by RAG context, the adapted prompt is present in Listing 4.2.

We directly prompt the zero shot prompt used in the previous experiment but we deleted the last part where is the information about PlantUML syntax due to token limitations and the fact that this guidance had already been included in the database. Looking at the result we could see that a class diagram was no different from the previously generated in the first experiment. Having similar results to the one presented by the Google model (Gemini).

To improve the quality of the generated diagrams, we rewrote the zero-shot prompt so that it was better suited for use in a RAG tool. The revised prompt included an explicit reference to RAG (Listing 4.2), direct instructions to leverage the knowledge base, and a more structured format to facilitate parsing. This adaptation produced better results than

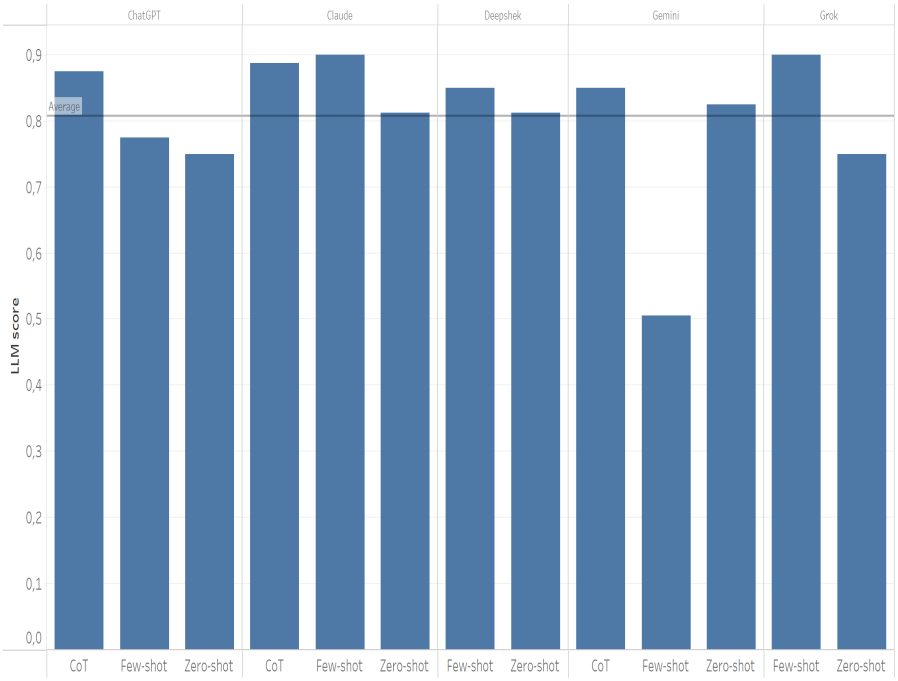


Figure 4.3: G-Eval of each LLM model per prompt technique.

the original prompt. The second diagram (Figure 4.9), not only identified an additional relationship absent from the first but also aligned more closely with the ground truth benchmark. However, in both cases the models struggled to capture all the necessary attributes. In particular, while the first diagram incorrectly represented the attributes Platform and MediaType as separate classes, reducing legibility, the second omitted these elements entirely, as seen in Figure 4.9.

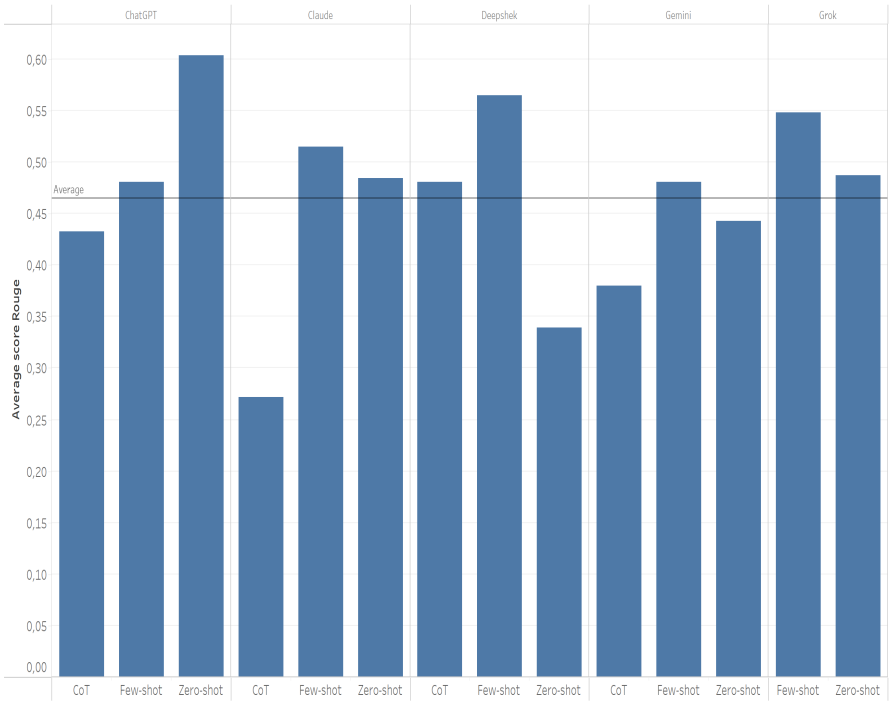


Figure 4.4: ROUGE of each LLM model per prompt technique.

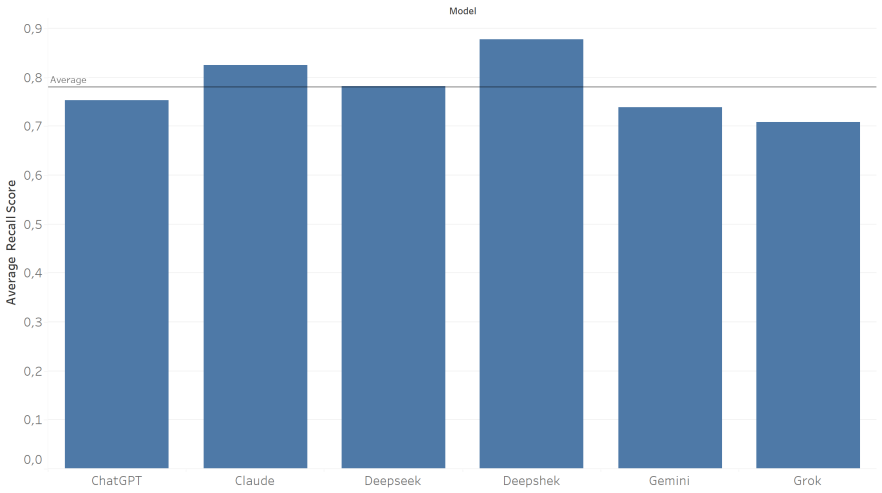


Figure 4.5: Recall of each LLM model overall.

```
You are an experienced software engineer specializing in UML modeling. Based on the UML
    specifications, PlantUML documentation, and modeling examples available in your
    knowledge base, create a class diagram for a project management system.

**System Requirements:**
The project manager leads a team to execute projects within defined start and end dates.
    Once created in the system, a manager may initiate and later terminate projects due
    to completion or other reasons. The team executes the project using requirements as
    input to produce a system as output.

Requirements and systems are work products with descriptions, completion percentages, and
    validation capabilities. Requirements are validated through user workshops, while
    systems are validated by testing against requirements. Requirements may be published
    via various media (intranet, paper), and systems may be deployed on specific
    platforms.
```

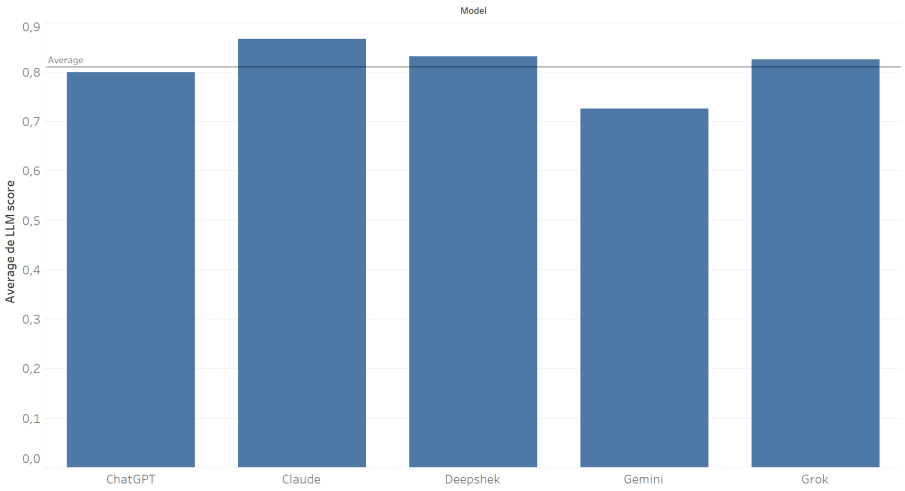


Figure 4.6: G-Eval of each LLM model overall.

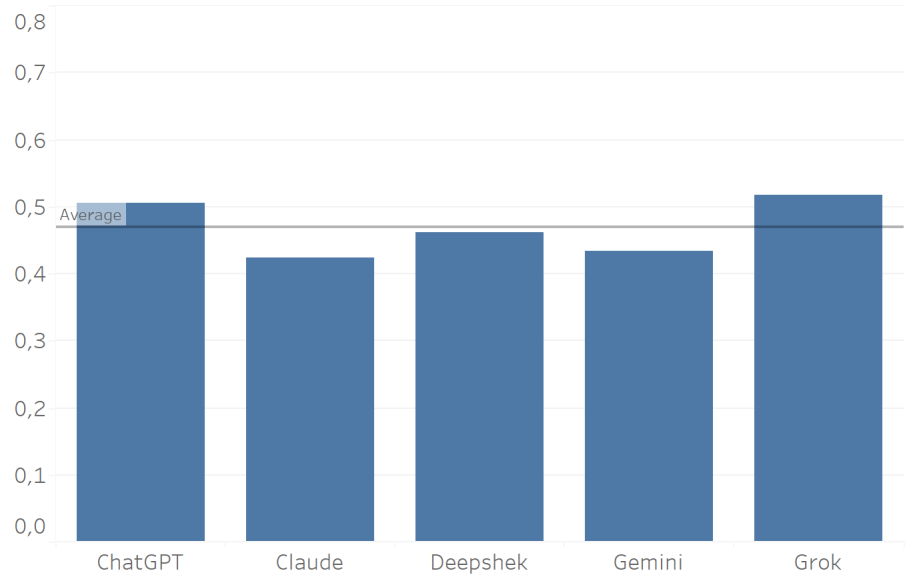


Figure 4.7: ROUGE of each LLM model overall.

4.2.2 Results and Discussion

Compared with the results obtained in the first experiment, the class diagram generated using the adapted prompt is the closest to the ground truth. It achieved the highest precision, recall, and F1-scores across all elements of the class diagram, with the exception of attributes. As shown in the UML class diagram in Figure 4.9, the use of an adapted prompt within the RAG framework improved the overall quality of the generated diagram by increasing the level of abstraction and simplifying the representation of the system, thereby enhancing readability. The consistently high precision and recall scores further indicate that the model effectively identified the relevant classes and relationships, underscoring the utility of RAG. Nevertheless, the lower performance on attributes suggests that,

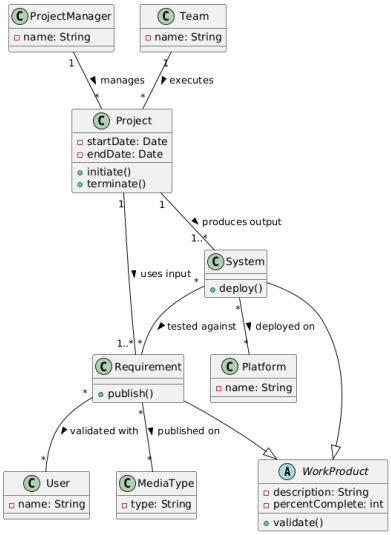


Figure 4.8: Zero-shot prompt

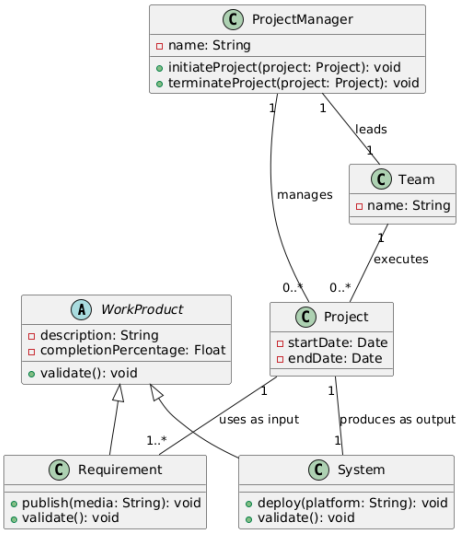


Figure 4.9: Zero-shot prompt adapted for RAG

while RAG contributes to structural accuracy, it may not sufficiently capture finer-grained details.

4.3 Experimenting Self-refinement

An emerging technique in the field of generative artificial intelligence is self-refinement, a process wherein LLMs iteratively improve their own outputs by generating feedback and revising responses without human intervention. This paradigm mirrors the structure of human-in-the-loop feedback systems but replaces the human evaluator with the model itself, thereby enabling a form of autonomous self-improvement. Madaan et al. (2023) formalized this approach through the Self-Refine framework, demonstrating that an LLM can act simultaneously as generator, critic, and refiner via prompt chaining. Their

experiments across diverse tasks—including mathematical reasoning and open-domain question answering—revealed an average absolute improvement of approximately 20% in performance compared to traditional one-shot generation methods. A key advantage of this technique is its ability to enhance model performance entirely at inference time, leveraging prompt engineering and in-context feedback, without resorting to computationally expensive fine-tuning or additional training [66]. This makes it particularly suitable for our setting with limited computational resources. Motivated by these promising results, we conducted a series of experiments to assess whether self-refinement could enhance the performance of our own generative system. Specifically, we aimed to explore the impact of iterative self-evaluation and refinement on the quality and accuracy of generated outputs within our use case.

4.3.1 Experimental Setup

To implement the self-refinement methodology, we developed a Python-based experimental pipeline that leverages the Anthropic Claude API to simulate a reflective reasoning loop. This loop enables a generative model to not only produce UML class diagrams from natural language system descriptions but also to autonomously evaluate and iteratively improve its own output based on structured critique. The Anthropic Claude model was selected due to its strong performance in prior experiments (see Section 4.1) and its free-tier access, which allowed us to execute multiple inference rounds without incurring computational or financial overhead. These practical and performance-based criteria rendered Claude an appropriate model for experimentation in resource-constrained environments. The self-refinement loop is implemented as a three-step iterative process capped at a maximum of three iterations. The sequence is as follows:

1. Initial Generation: Firstly, the system uses a chain-of-thought (CoT) prompt, previously evaluated and available in Appendix C, to generate the class diagram based system description. This response constitutes the starting output to be refined.

2. Self-Evaluation: The generated diagram is passed to a structured evaluation function which queries the Claude model to assess the quality of the diagram across four rigorously defined dimensions: UML completeness, **UML syntax and quality, requirement coverage, and clarity and pragmatics.** Each dimension is weighted, and an overall evaluation score is computed. The response includes both scalar scores and detailed natural language feedback in JSON format, ensuring both interpretability and machine-readability. The prompt can be found in Appendix C.

3. Self-Improvement: If the initial output does not meet the quality threshold—defined or higher a refinement step is triggered. In this step, the previously generated diagram, the system description, and the evaluation feedback are incorporated into an improvement prompt (Listing C.2). This prompt instructs the model to revise the class diagram to resolve the identified issues and implement the suggested

enhancements. The revised output is then subjected to a new round of self-evaluation. The prompt can be found in Appendix C.

This loop continues iteratively until either a satisfactory score is achieved or the maximum number of iterations is reached. To support transparency and replicability, the pipeline maintains a persistent logging system that records all iterations, feedback evaluations, and diagram versions. Additionally, a structured feedback_history data structure is maintained, which stores for each iteration: (i) the feedback JSON object, (ii) the corresponding PlantUML diagram, and (iii) the iteration number. This facilitates post-hoc analysis and supports visual inspection of improvement trends. Overall, this experimental design allows us to rigorously test whether self-reflective generation—achieved entirely through inference-time prompting—can lead to substantial and consistent improvements in UML modeling tasks. The design aligns with emerging research advocating for *low-cost, feedback-driven refinement frameworks* (e.g., Madaan et al., 2023), which offer a viable alternative to expensive fine-tuning or supervised retraining in downstream applications.

To decide when to stop the loop, we used a threshold. To know which is the best threshold that fits our goal the better we run the script with different thresholds ranging from 0.7 to 0.95. As it can be seen in the Figure 4.10, we recorded the average score, average iterations done and the standard deviation of the score. This way we can analyse the effort needed to achieve a certain threshold and the stability of those scores. The thresholds 0.75 to 0.85 have high average scores and low iteration scores. The threshold 0.85 seems to be more stable. The threshold 0.9 achieves the highest scores but at the cost of the number of iterations. Also, it has the highest standard deviation which indicates that is less predictable in terms of performance. The last threshold has slightly worst score then the previous threshold but with a higher number of iterations. Although is more stable then 0.9 threshold. Looking at this results we end up deciding to use 0.85 because it offers the best cost-benefit ratio.

4.3.2 Results and Discussion

The self-improvement pipeline was executed on the same system description used in the previous experiments. After a single iteration the score grew from **0.815** to **0.855**.

Figures figs. 4.11 and 4.12 show, respectively, the diagram produced in the first pass and the diagram obtained after one refinement cycle. While the initial model already contained the eight core domain classes, their relationships and every public operation required by the ground-truth reference, the refinement step enriched the artefact in several important ways that are directly visible when the two diagrams are compared, that goes beyond what is expressed in the system description:

- **New domain concept.** A dedicated TeamMember class now appears, linked to Team through a composition (1 `Team` --> 1.. * `TeamMember`). This makes individual contributors first-class citizens of the model.

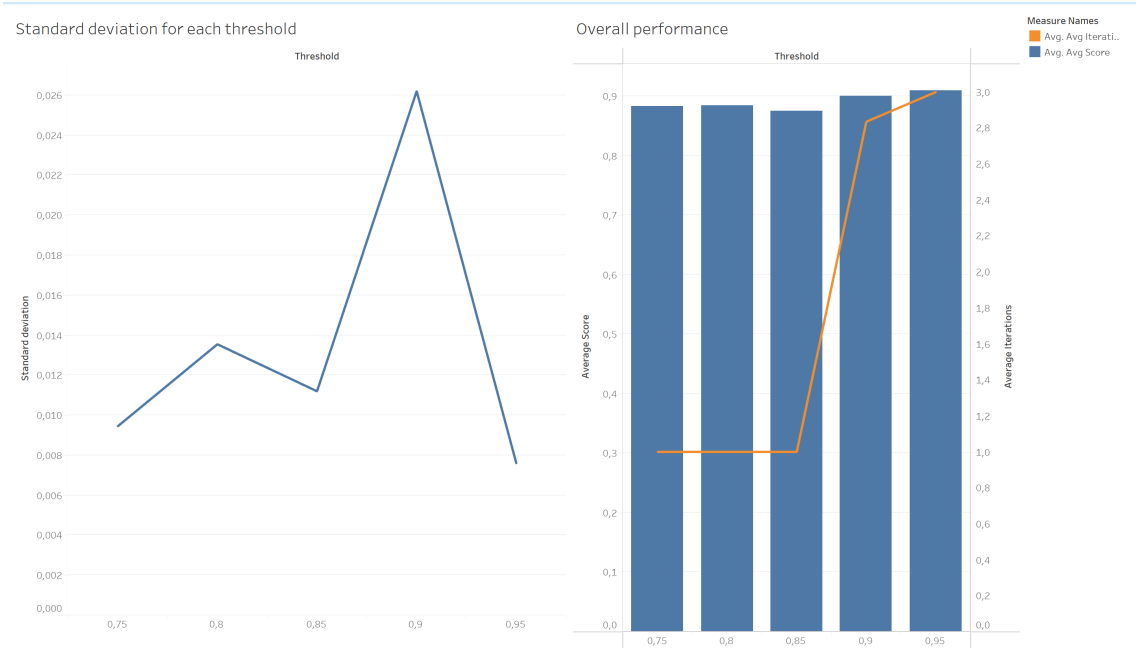


Figure 4.10: Threshold hold test. The blue bars represent the Average Score (left), while the orange line represents the (right).

- **Richer team management behaviour.** The Team class gained a name attribute together with membership-handling services (`addMember`, `removeMember`, `getMembers`). Complementarily, ProjectManager now offers `assignTeam(Project, Team)`, and TeamMember provides assignToTeam, removeFromTeam and addSkill.
- **Explicit traceability between artefacts and validation.** WorkProduct now maintains a collection `validations: List<Validation>` and is connected to Validation through an association labelled *validated by* with cardinalities $1 \leftrightarrow 0..*$. Corresponding operations (`addValidation`, `getValidations`) were also added.
- **Refined requirement compliance.** A new association *validates against* relates System to one or more Requirement instances ($1 \leftrightarrow 1..*$), turning the previously implicit link embodied in testAgainstRequirements into an explicit structural dependency.
- **Enhanced project coordination.** The Project class is now able to register requirements and the target system by means of the newly introduced operations addRequirement and setSystem.
- **Extended validation feedback.** Validation itself now exposes accessor methods getValidationResult and getValidationDate, providing programmatic access to the outcome and time-stamp of the validation activity.

Overall, the iteration increased the number of associations, attributes and operations, thereby improving model completeness without sacrificing correctness: no previously correct element was removed or altered erroneously. Although the quality when compared

to the ground truth did not increased, there is an increase on the details which is important for following ativities after modelling.

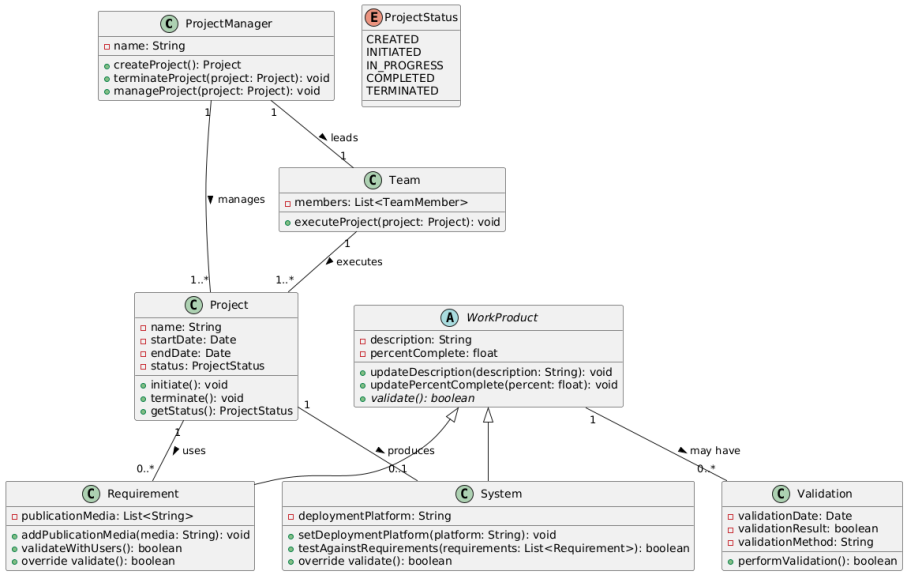


Figure 4.11: Initial diagram.

4.4 Summary

This chapter presents a comparative study of techniques for generating UML class diagrams from natural language system descriptions using large language models (LLMs). The experiments systematically evaluate the impact of different prompt strategies (zero-shot, few-shot, and chain-of-thought) and model choices (ChatGPT-4o, Claude Sonnet 4, DeepSeek XL, Gemini 2.5 Flash, and Grok 1.5) based on structural, semantic, and lexical criteria. The main findings from the experiments are summarised below:

- **Prompt strategies**
  - Chain-of-thought prompting improved recall and semantic coverage, particularly for low-salience elements.
  - Few-shot prompting achieved the highest ROUGE scores, indicating stronger lexical alignment.
- **Model comparison**
  - Claude Sonnet 4 was the most balanced model in terms of structural and semantic performance.
  - Open-source models were excluded from further analysis due to weak results and practical constraints.
- **Retrieval-Augmented Generation (RAG)**

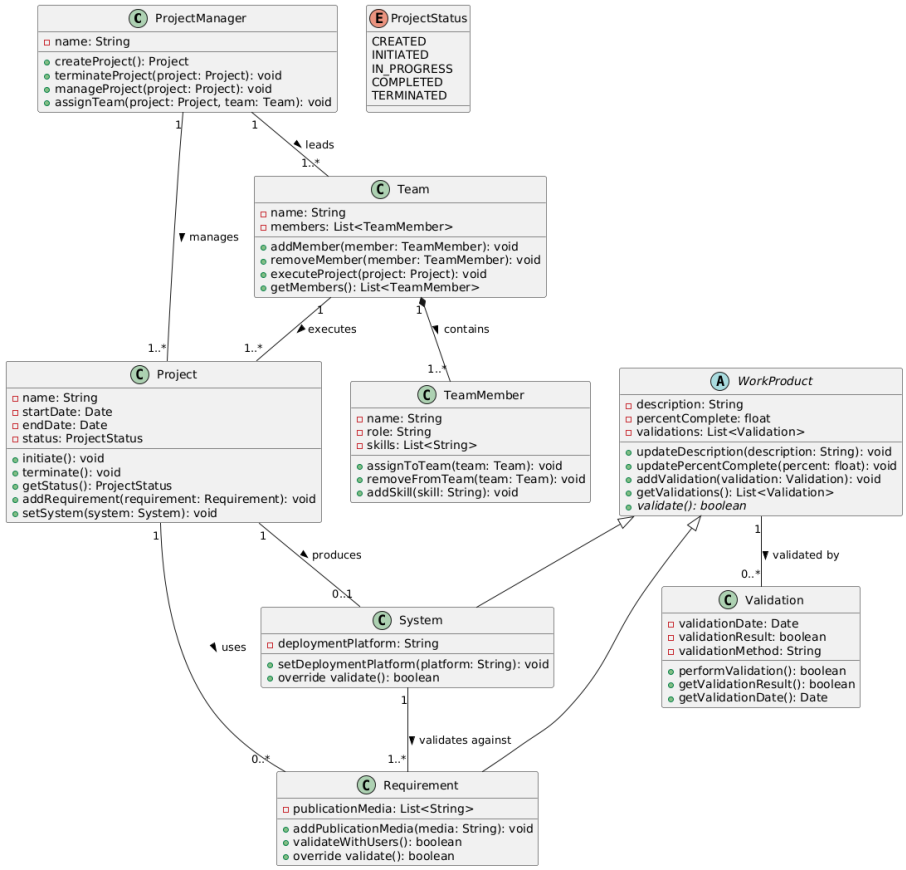


Figure 4.12: Diagram after one self-improvement iteration.

  - In NotebookLM, an adapted zero-shot prompt that leveraged document-grounded context produced diagrams closest to the ground truth, outperforming both standard zero-shot and few-shot setups.

- **Self-refinement (Claude + Self-Refine)**
  - The system autonomously generated, critiqued, and iteratively refined its UML output until a predefined quality threshold was met.
  - This process increased completeness and detail—adding relevant classes and relationships—without reducing correctness.
  - Threshold testing indicated that a target score of **0.85** offered the best trade-off between quality and efficiency.

Our approach uses a, the combination of prompt engineering, document grounding, and self-refinement to provides a low-cost, effective strategy for improving the quality of UML diagram generation in resource-constrained environments.

5

An approach to automate the creation of class diagrams

This chapter presents our approach to automate the creation of UML class diagrams from natural language system descriptions. Building on the experiments reported in the previous chapter, we evaluate the performance of different Anthropic models with the CoT prompting, comparing recall, cost-performance, and overall effectiveness in the task. Based on these insights, we select the most suitable models to support our approach. We then describe the design and implementation of a RAG system. This includes an overview of the components tested, the evaluation procedure, and the metrics used to identify the best configuration for our pipeline. Next, we detail the self-refinement loop implemented, explaining its structure, the prompts designed for each stage, and the stopping criteria established to balance performance improvements with cost efficiency. Finally, we present the overall system architecture, which integrates the RAG pipeline and the self-refinement loop into a unified process. We discuss the main challenges encountered during development, particularly cost management and RAG configuration, and conclude with a practical demonstration that provides a practical demonstration of the system applied to a realistic use case, comparing the AI-generated class diagram to the ground truth and drawing conclusions from the results.

5.1 Prompt technique and Model used

From the experiment presented in the previous chapter (Chapter 4), we observed that Claude offered the most balanced performance, achieving the best G-eval results. Additionally, models provided by Anthropic have demonstrated strong results in software engineering tasks, as evidenced by the SWE-Leaderboard[1]. Consequently, we decided to use CoT prompting, which achieved the best overall results as shown in Section 4.1. Having selected the prompt strategy and model family, we needed to determine the specific Claude model to employ. Anthropic's API offers several models within the Claude family,

[1] https://swe-bench-live.github.io/

each optimized for different performance characteristics. The Claude 4 family includes Claude Opus 4, designed for complex reasoning tasks requiring deep analytical capabilities, and Claude Sonnet 4, which provides a balance between intelligence and efficiency suitable for a wide range of applications. Earlier generations, such as Claude 3.5 Sonnet and Claude 3.5 Haiku, remain available with varying trade-offs between reasoning depth, processing speed, and computational cost. These models share core capabilities including large context windows, strong performance on coding tasks, and robust safety features, while differing primarily in their level of sophistication and resource requirements. To inform our model selection, we conducted an experiment using the same experimental setup employed in the previous chapter to evaluate the different Claude models available through the Anthropic API.

From the results we can see that the models faced difficulties in identifying the attributes and the methods present in the ground truth resulting in a lower recall value (as it can be seen in Figure 5.1). This can be due to the system description not being very clear in some attributes and methods which reconfirms the importance of complete unambiguous system description as it was expressed by Ruan et al. and Chaaben et al. [7, 47].

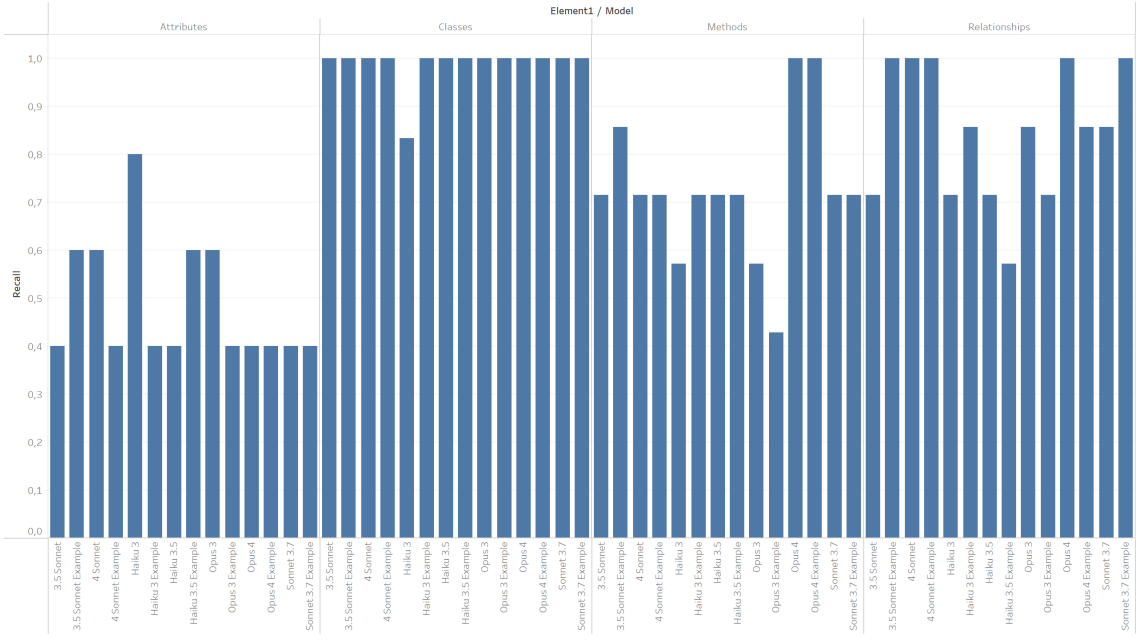


Figure 5.1: Recall per element of the class diagram

Since we are using Anthropic API, each call is priced based on the number of input tokens and output tokens per one million tokens. To calculate the price we multiply the price per million tokens by the number of tokens in both the input and output. Consequently, the cost increases with the number of input tokens. Considering this, we tested the impact of adding examples of class diagram creation based on system descriptions to the prompt.

To better evaluate the overall performance of the models we calculated a overall score.

$$\begin{aligned}\text{Overall Score} = \frac{1}{4}\Bigg(&\frac{\text{Validity Score} + \text{Completeness Score}}{2} \\ &+ \frac{\text{Pragmatic Score} + \text{Pragmatic Clarity Score}}{2} \\ &+ \frac{\text{Syntactic Score} + \text{UML Completeness Score} + \text{UML Quality Score}}{3} \\ &+ \text{Requirement Coverage Score}\Bigg)\end{aligned} \tag{5.1}$$

Looking at the overall score we can see that out of the six models, five benefit from having an example in the prompt. reinforcing the hypotheses that adding examples to the prompt can improve the result of the LLMs, the results can be seen in the Figure 5.2.

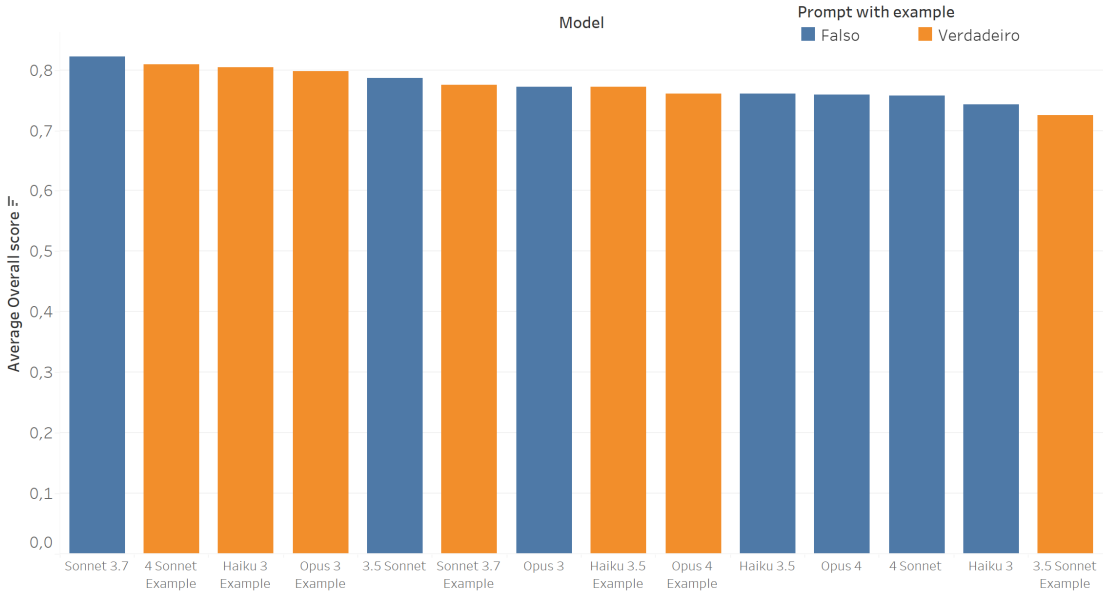


Figure 5.2: Overall score of the models with and without example in the prompt

We aim to have all the elements present in ground truth diagrams. Therefore, it is important that the model that we will be using has a good recall, meaning that all elements are represented in the class diagram generated. Looking at the recall average for each model, we can see that the best one is 3.5 Sonnet followed by the most powerful models Opus 4 and Sonnet 4. Besides these three models, 3.7 Sonnet also presents good results.

When deciding the model, it's also important to have in consideration the relations cost-performance, to do so we calculated the prices of each prompt based on the number of input tokens we were sending and the max output tokens established (2000 tokens).

As seen in Figure 5.3, the best cost-performance model would be present in the second quadrant. There are three models in this quadrant (Sonnet 3.7, Sonnet 3.5 and Haiku 3). Out of the three, Sonnet 3.7 has the best score. For our use case the top two models would be 3.7 Sonnet and 3.5 Sonnet. The Sonnet 3.7 had better overall result then Sonnet 3.5, however Sonnet 3.5 had better F1-score and recall and is cheaper that Sonnet 3.7 per call.

Inclusively, Haiku 3 is also a good option since it has a good cost-performance and is the cheapest model.

In conclusion, we will be using ,in the final implementation, the model 3.7 Sonnet and 3.5 Sonnet, but in the development phase we will be using Haiku 3 since it is the cheapest model and will allow us to reduce the costs during the development phase.

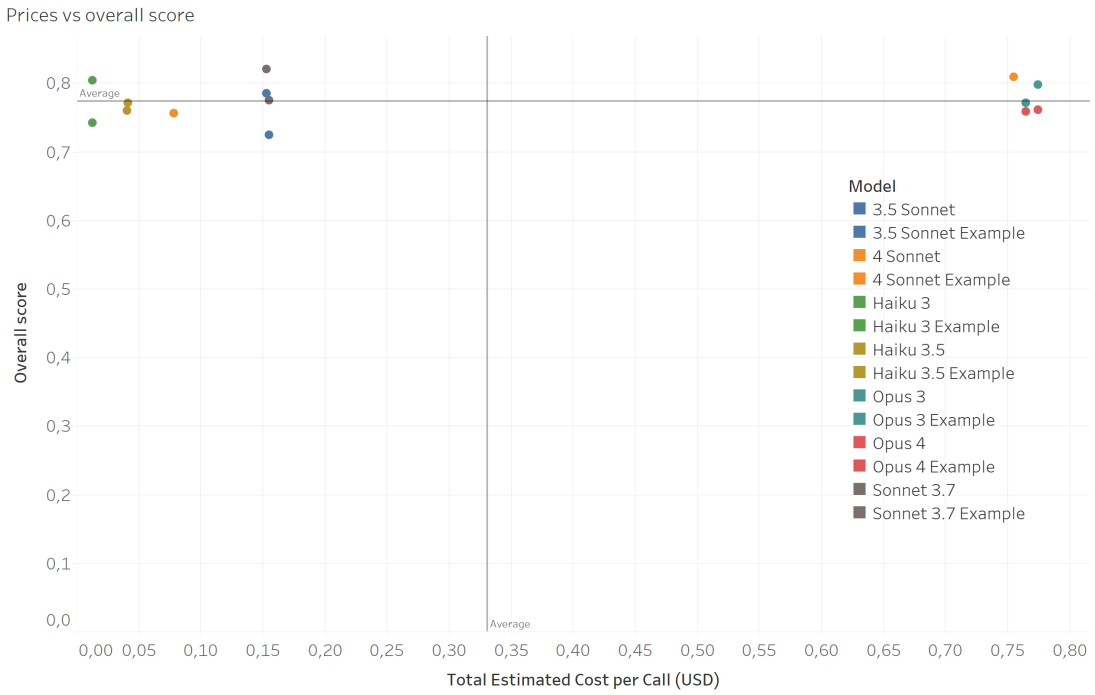


Figure 5.3: Overall score vs price of the models

5.2 Design and Implementation of the Retrieval-Augmented Generation System

After selecting the model and the prompt, we proceeded to design and configure our Retrieval-Augmented Generation (RAG) system. Due to time and hardware constraints, we were only able to test a limited number of component combinations. To ensure feasibility, we focused on **simple and lightweight configurations** that could run efficiently on local infrastructure.

To support this process, we developed a custom script to automate the creation and evaluation of different RAG pipeline configurations. Each configuration was composed by combining embedding models, text splitters, retrievers, and a vector store.

The different components we tested were:

1. **Embedding models**: models that convert text into dense vector representations enabling similarity search;

a) all-MiniLM-L6-v2: maps sentences and paragraphs to a **384 dimensional dense vector space** and is used for tasks such as clustering or semantic search;

b) all-mpnet-base-v2: maps sentences and paragraphs to a **768 dimensional dense vector space** and is used for tasks such as clustering or semantic search;

c) multi-qa-MiniLM-L6-cos-v1: maps sentences and paragraphs to a **384 dimensional dense vector space** and is optimized for **question-answering tasks** by being trained on 215M (question, answer) pairs from diverse sources;

d) all-distilroberta-v1: maps sentences and paragraphs to a **768 dimensional dense vector space** and is used for tasks such as clustering or semantic search, it was used the pretrained distilroberta-base model and fine-tuned in on a 1B sentences in on a 1B sentences pairs dataset;

e) paraphrase-multilingual-MiniLM-L12-v2: maps sentences and paragraphs to a **384 dimensional dense vector space** and is used for tasks such as clustering or semantic search, it was trained on a large corpus of multilingual data to improve its performance on non-English text.

2. **Text splitters**: Strategies to divide documents into smaller chunks to facilitate storage and improve efficiency;

a) Recursive text splitter with chunk size = 300 tokens and chunk overlap = 20 tokens - Default configuration, balancing context and efficiency;

b) Recursive text splitter with chunk size = 1024 tokens and chunk overlap = 200 tokens - Large Chunks to preserver a broader context;

c) NLTK text splitter with chunk size = 300 tokens and chunk overlap = 20 tokens - Uses linguistic rules for more natural splits.

3. **Retrievers** - Algorithms to decide which chunks are more relevant for a given query;

a) Similarity with $k = 3$ - Retrieves the top 3 most similar chunks by cosine similarity;

b) MMR with $k = 3$ and fetch_k = 10 - Maximal Marginal Relevance to balance relevance and diversity;

c) Similarity with a score threshold = 0.5 and $k = 10$ - retrieves the top 10 chunks with similarity score above 0.5;

d) BM25 - information retrieval based on term frequency and document length;

e) Ensemble (0.5 BM25 + 0.5 vectorstore) - method that combines traditional retrieval techniques with embedding-based retrieval.

4. **Vector store** - database for storing and retrieving vector embeddings;

a) Chroma - open-source vector store designed for local and efficient RAG experiments.

To evaluate which the best configuration is, we created a python script that would create all the possible combinations of the components mentioned above, creating a total of 60 configurations. Each configuration was tested using a set of 15 open-ended questions related to UML, PlantUML. The questions were designed to cover a range of difficulty levels, from basic definitions to complex scenarios involving multiple concepts. Each question has an expected answer and key concepts associated with it. Each question was used as a query for the RAG system, and the results were evaluated based on the metrics mentioned above. Here is how each metric was calculated:

- **Retrieval time**: Time taken to retrieve relevant chunks of information from the documents based on the query;
- **Number of chunks generated**: Total number of chunks created from the documents when storing them in the vector store;
- **Accuracy**: Calculated using keyword matching and semantic similarity between the expected answer and the response generated by the RAG system; it reflects the degree to which the retrieved information is correct and related to the query;
- **Semantic similarity**: Calculated using cosine similarity between the embedding of the expected answer and the embedding of the response generated by the RAG system; it expresses the extent to which the retrieved chunks are semantically related to the query;
- **Completeness**: Calculated based on the presence of all key concepts in the response generated by the RAG system; it measures the degree to which all relevant information for answering the query is included in the retrieved chunks;
- **Relevance**: Calculated based on the arithmetic mean of the similarity scores between the chunks retrieved and the query; it reflects the extent to which the retrieved chunks are relevant to the query;
- **Syntax correctness**: Calculated using regex patterns to check for the PlantUML syntax elements (such as relationship declarations, stereotypes, start and end markers) that are present in both the expected answer and the response generated by the RAG system; it indicates the degree to which the response adheres to UML syntax rules.

For an overall evaluation, we computed an overall score for each configuration using the formula 5.2 . This score provides a balanced view of the configuration's performance across all metrics.

$$\begin{aligned}\text{Score} = \big(&0.25 \times \text{Accuracy} + 0.25 \times \text{Semantic Similarity} + 0.20 \times \text{Completeness} \\ &+ 0.15 \times \text{Relevance} + 0.15 \times \text{Syntax Correctness}\big)\end{aligned} \tag{5.2}$$

5.2.1 Results and Discussion

Looking at the top 10 overall, all had a score higher than 0.67318, although the difference between top and bottom configurations is statistically small, as illustrated in Figure 5.4. This suggests **diminishing returns** and points to the importance of secondary metrics (e.g., speed, semantic quality) when making trade-offs.

The best overall configuration is:

- Embedding: `multi-qa-MiniLM-L6-cos-v1`
- Splitter: Recursive (chunk size = 300, overlap = 20)
- Retriever: MMR (k = 3, fetch_k = 10)

This setup achieved:

- Overall score ≈ 0.684
- Retrieval time ≈ 0.044 s
- Semantic similarity ≈ 0.567
- Accuracy ≈ 0.631
- Relevance ≈ 0.593
- Chunks: 2580

While it produced the **highest number of chunks**, this was expected due to the small chunk size (300). The higher chunk count contributed positively to retrieval quality, particularly relevance and completeness.

In terms of **semantic similarity**, this configuration ranked 5th among the top 10, as seen in Figure 5.5. However, since **semantic similarity and relevance showed no linear correlation** (Pearson $r \approx 0.02$), this metric should be interpreted independently.

The configuration achieved **second-highest accuracy** among top-ranked results, confirming its robustness across both semantic and factual dimensions.

Regarding **retrieval time**, this setup was not the absolute fastest but remained well within efficient limits (< 0.05 s). The **fastest configuration** used `all-MiniLM-L6-v2` with MMR and the NLTK splitter, achieving ≈ 0.15 s, but with a slightly lower overall score.

Chunk count analysis showed that smaller chunk sizes (e.g., 300) improved performance at the cost of retrieval volume, whereas larger chunks did not improve scores despite reducing memory and time costs.

MMR-based retrievers mostly outperformed others, especially when paired with MiniLM or multi-qa embeddings.

`multi-qa-MiniLM-L6-cos-v1` and `all-MiniLM-L6-v2` emerged as the best overall embedding models, appearing repeatedly in the top 3 combinations.

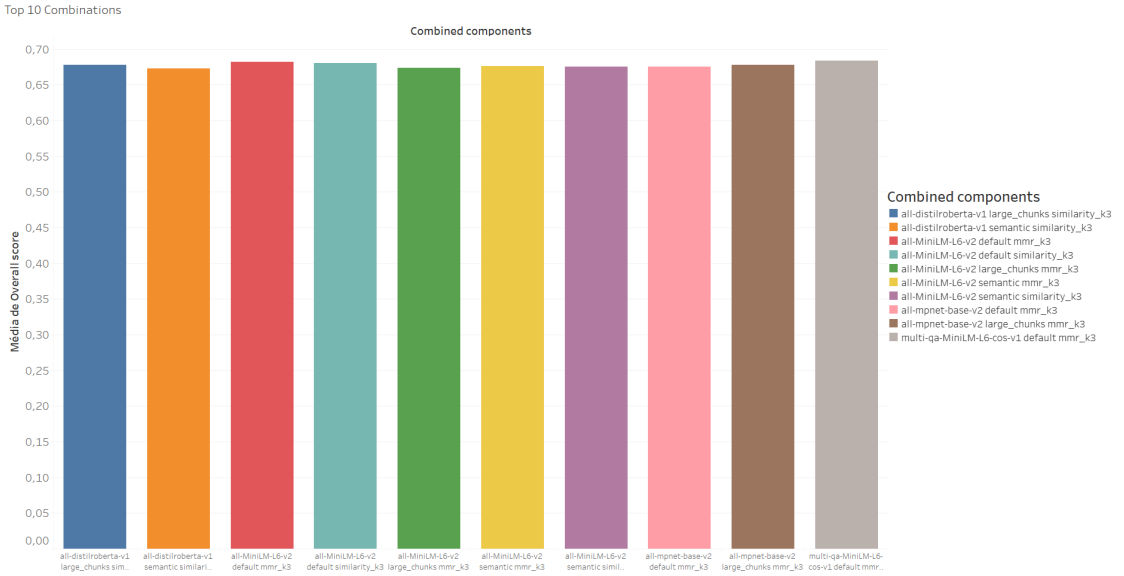


Figure 5.4: Top 10 configurations

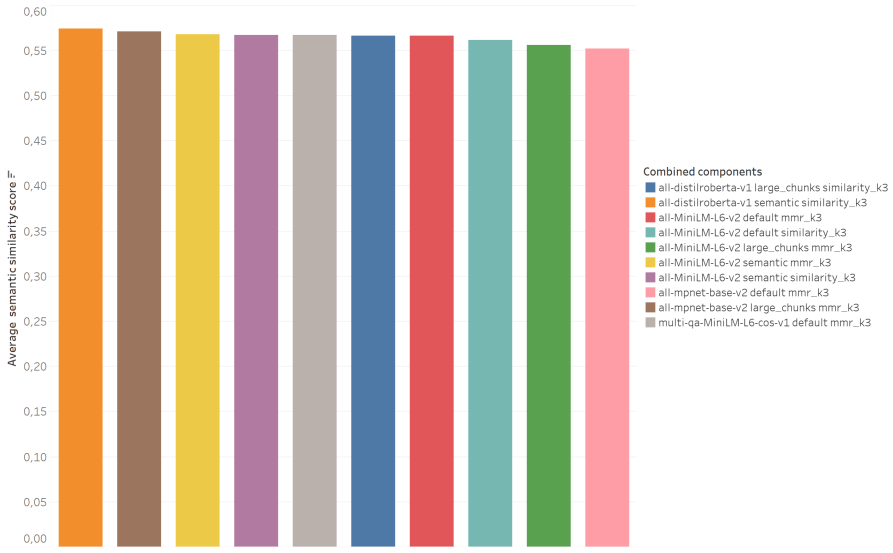


Figure 5.5: Semantic similarity score of the top 10 configuration

Retrieval time and relevance showed a **moderate positive correlation**, as it can be seen in Figure 5.6 — slower retrievals tended to return more relevant chunks, suggesting a trade-off worth tuning based on latency needs.

The benchmark shows that the configuration using `multi-qa-MiniLM-L6-cos-v1`, the default recursive splitter (chunk size = 300 tokens), and MMR (k = 3, fetch_k = 10) delivers the **most balanced performance**, combining strong relevance, accuracy, and efficiency.

Despite small differences in overall scores among the top 10, this configuration stands out due to its ability to scale with a low retrieval time, high semantic coherence, and robust accuracy.

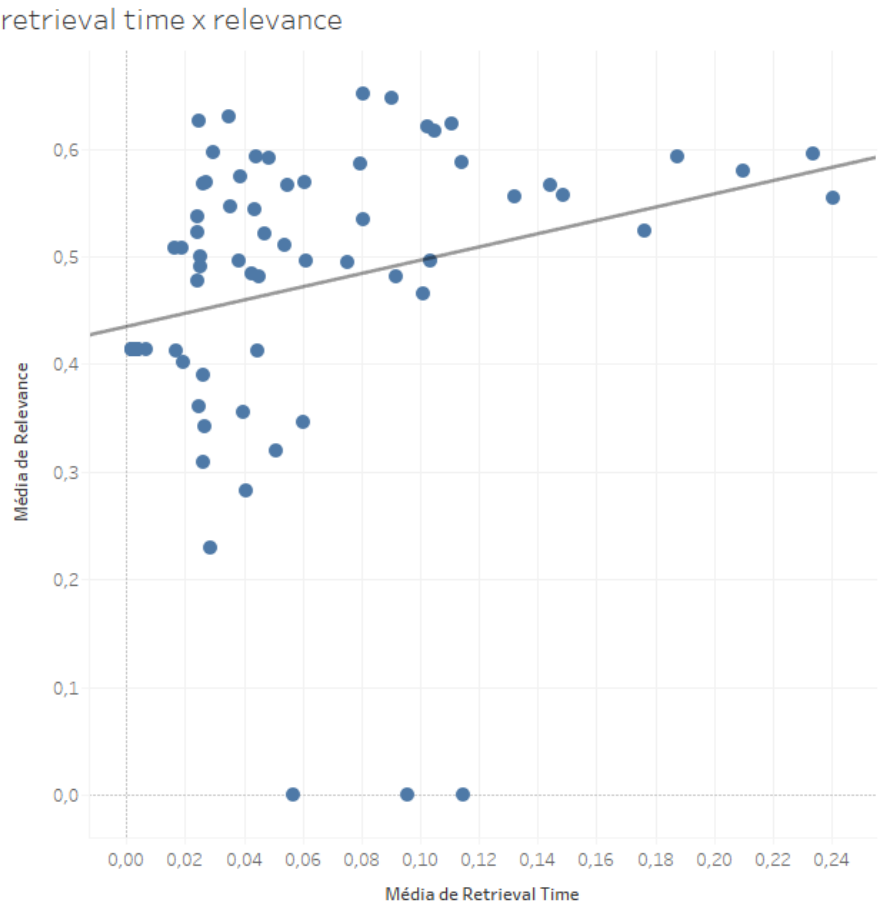


Figure 5.6: Retrieval time and relevance correlation

5.3 Self-refinement approach

Inspired by the iterative refinement framework proposed by Madaan et al. [66], this study implements a self-refinement loop in which the same large language model (LLM) is responsible for **generating**, **evaluating**, and **refining** UML class diagrams in an autonomous and cyclical manner. Figure 5.7 illustrates the self-refinement loop.

To prevent excessive or redundant refinement, we impose an upper bound of three iterations per cycle. This decision is empirically supported by the findings of Madaan et al. [66], who observed that while performance improves substantially during the initial two iterations, subsequent iterations tend to yield diminishing returns in both quality and novelty. In addition to this upper limit, we also introduce a **dynamic stopping condition**: if the improvement in the evaluation score between the current and the previous iteration falls below a threshold of **0.01**, the loop is terminated early. This ensures that API calls are cost-efficient and only meaningful refinements are performed.

Furthermore, to enhance consistency and reduce redundancy across iterations, we incorporate **conversational memory** into the prompt context. The model is provided

with the content of the two most recent interactions (both the prompt and the corresponding output). This addition was empirically motivated: during initial experimentation, we observed that including minimal conversational history significantly improved the model's ability to avoid previously made errors and reduced the likelihood of introducing redundant or contradictory modifications. However, this improvement in performance comes at the cost of a slightly increased token consumption per API call.

To structure the interactions with the model throughout the self-refinement loop, we adopt a **CoT prompting strategy** [67], which encourages the model to reason through tasks step by step. Each of the three system prompts, **generation**, **evaluation**, and **improvement** was carefully engineered to align with the cognitive role expected of the model during its respective task.

- The **generation prompt** is framed in a descriptive and procedural tone. It guides the model step-by-step through the tasks of entity identification, relationship analysis, attribute extraction, application of UML design principles, and final validation (can be seen in Listing C.1).

- The **evaluation prompt** adopts a formal and analytical tone, simulating the role of an academic reviewer or UML expert. It instructs the model to assess the class diagram against a rubric of weighted criteria (can be seen in Listing C.2).

- The **refinement prompt** takes on a mentor-like tone, instructing the model to act as a senior software architect improving a junior designer's work (can be seen in Listing C.3).

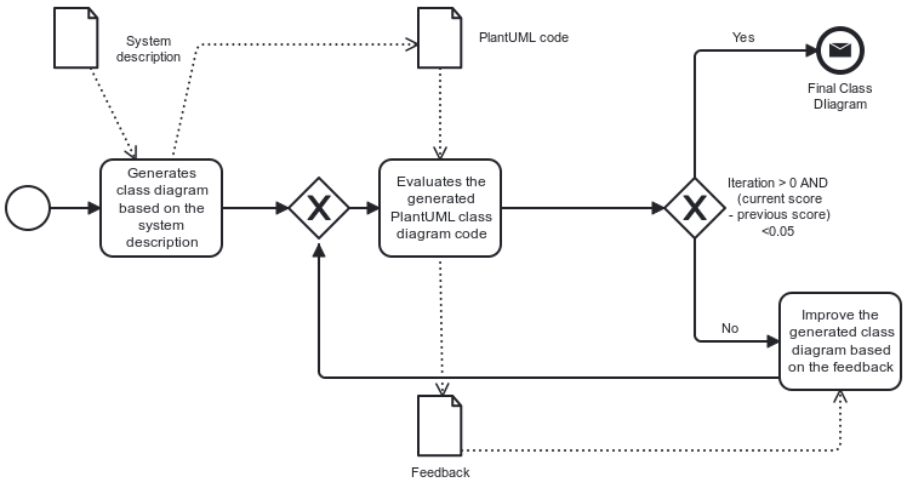


Figure 5.7: Self-refinement loop architecture.

5.4 Automatic class diagram creation pipeline using system description

To operationalize the pipeline, we integrate all developed components into a coherent architecture. As illustrated in Figure 5.8, the system is composed of three main modules, each responsible for a distinct phase of the process. With all components developed, we now integrate them into a unified system. As illustrated in Figure 5.8, the architecture is composed of three main modules:

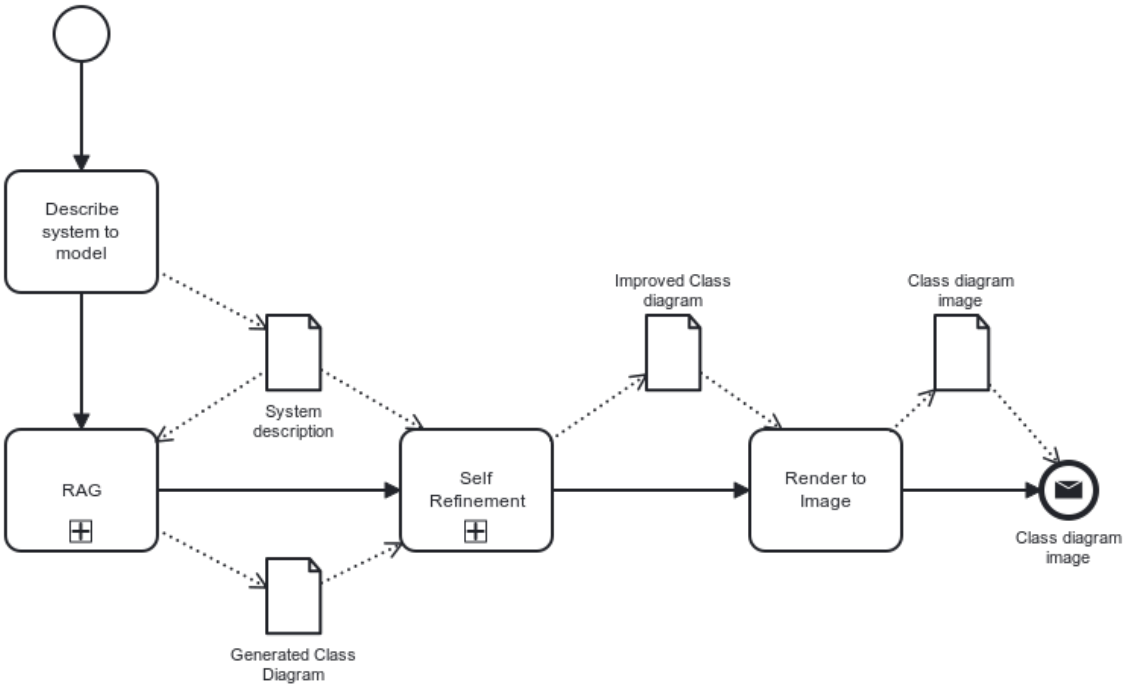


Figure 5.8: Overall architecture of the proposed system.

1. **Generation Module** - Responsible for producing the initial class diagram using a RAG system;
2. **Refinement Module** - Refines the initial diagram based on self-evaluation;
3. **Rendering Module** - Converts the PlantUML code into a PNG image for visualization;

For all calls to Anthropic's Claude Sonnet, we fixed the decoding temperature to T=0.2, in order to reduce randomness and ensure that differences across conditions are attributable primarily to prompt strategy rather than sampling noise. Beyond controlling for sampling variability, we also refined our knowledge retrieval strategy. In the final implementation, we adopt a distinctive RAG approach that decouples the retrieval query from the user's input. Instead of directly using the system description as the query for retrieval, we utilize a predefined set of domain-specific questions to extract relevant knowledge chunks from

our database. For example, when a user intends to create a class diagram, the predefined questions are:

- *"What is the PlantUML syntax for drawing class diagrams?"*;
- *"What are the best practices for UML class diagrams?"*;
- *"What are the common mistakes in UML class diagrams?"*;
- *"What are the key principles of UML class diagrams?"*.

These questions, along with the retrieved content, are passed to the Haiku 3 model, which generates natural language responses constrained to a maximum of 1000 tokens. This way we guarantee that the output remain concise and not overly verbose. These responses then serve as contextual input for the prompt that generates the initial class diagram from the system description. This modular and context-enriched strategy is extensible to other types of UML diagrams beyond class diagrams.

Once the PlantUML code for the class diagram is generated, the system proceeds to refine it. The refinement process follows the methodology and prompt design described in the previous section, leveraging self-evaluation and iterative refinement strategies.

After refinement, the final PlantUML code is saved with a unique identifier and rendered into a PNG image using the official PlantUML JAR tool. The resulting diagram is then presented to the user via the interface.

5.5 Challenges

This section details the two main challenges encountered during the development of the system. The operational costs associated with LLM API usage and the complexity of selecting and configuring RAG components are discussed. For cost reduction we apply mitigation strategies such as prompt optimization and output structuring. For RAG configuration, we adopt a pragmatic approach focusing on simplicity and empirical evaluation of common tools.

5.5.1 Cost Reduction in LLM API Usage

The use of commercial Large Language Model (LLM) APIs, such as Anthropic's Claude models, entails operational costs. These are typically calculated based on the number of input and output tokens per request, with pricing varying by model (e.g., Claude 3 Haiku vs. Claude 3 Opus).

To balance performance and cost-efficiency, we conducted preliminary experiments (see "Prompt Technique and Model Used" section) to identify a model that provided satisfactory results while maintaining a lower cost per call. Haiku 3 and 3.7 Sonnet emerged as viable options due to their relatively low pricing and competitive performance.

In addition, we implemented several mitigation strategies:

- **Prompt Optimization**: Prompts were crafted to be concise, retaining only essential information. This reduced the input token count while preserving effectiveness.

- **Structured Output Specification**: We explicitly defined the desired output format (e.g., JSON or PlantUML) within the prompt. This constrained the response length and improved consistency, which in turn minimized the number of output tokens.

These techniques collectively contributed to the reduction of total API usage costs without significantly compromising generation quality.

5.5.2 Selecting and Configuring RAG Components

The construction of a RAG system involves a wide variety of configurable components, including chunking strategies, embedding models, retrievers, and re-ranking methods. Given the combinatorial nature of these options, identifying the optimal configuration posed a significant challenge. To manage this complexity, we adopted a pragmatic approach:

- **Scope Limitation**: Rather than testing the full spectrum of RAG architectures, we focused on implementing a basic (or "naive") RAG pipeline.

- **Use of Open-Source Tools**: We prioritized open-source and freely available components to ensure cost-efficiency and reproducibility.

- **Empirical Evaluation**: From the pool of commonly adopted tools (e.g., HuggingFace Transformers, FAISS, and Chroma), we selected the most widely used and tested them across different configurations, as it was detailed in the Section 5.2.

By narrowing the scope and selecting community-supported tools, we were able to reduce implementation complexity while still conducting meaningful evaluations.

5.6 Practical Demonstration

In this section we present our proposed system and show its feasibility by applying it to a realistic use case, demonstrating how our automatic process can be applied to a system description and examining the results.

For this proof of concept, we used a system description of a Patient Record and Scheduling system (Listing 5.1) present in the dataset [59].

```
A patient record and scheduling system in a doctor's office is used by the receptionists,
    nurses, and doctors. The receptionists use the system to enter new patient
    information when first-time patients visit the doctor. They also schedule all
    appointments. The nurses use the system to keep track of the results of each visit
    including diagnosis and medications. For each visit, free form text fields are used
    to capture information on diagnosis and treatment. Multiple medications may be
    prescribed during each visit. The nurses can also access the information to print
    out a history of patient visits. The doctors primarily use the system to view
    patient history. The doctors may enter some patient treatment information and
    prescriptions occasionally, but most frequently they let the nurses enter this
    information. -- Each patient is assigned to a family. The head of family is
    responsible for the person with the primary medical coverage. Information about
    doctors is maintained since a family has a primary care physician, but different
    doctors may be the ones seeing the patient during the visit.
```

Listing 5.1: System Description: Patient Record and Scheduling System

For a clear understanding of how the transformation approach works, the following steps, illustrated through the interface shown in Figure 5.9, describe the sequence of events leading to the final diagram:

1. The user selects the type of diagram to generate.
2. The user describes the system in the text box.
3. The user clicks on the button to generate the diagram.
4. The frontend sends a request to generate the diagram.
5. The user waits between 30 and 60 seconds.
6. After generation, the user can save the image and view the corresponding PlantUML code.

The result can be seen in Figure 5.10, and it is compared to the ground truth diagram in Figure 5.11.

The generated class diagram includes the set of core domain concepts (Patient, Visit, Appointment, Doctor, Medication and Family) present in the system description. Besides these core concepts, also identified in the ground truth, it includes classes to represent the nurse workers and the receptionists, both mentioned in the system description and associated with functional requirements.

It introduces an abstract Person class, which Patient and Receptionist inherit from. This ensures that shared details such as names and contact information are only defined once, avoiding duplication and making the design easier to maintain. It also adds more specific roles like Nurse and Receptionist, which were not included in the ground truth diagram. This helps to more accurately model the actual roles and responsibilities within a healthcare setting.

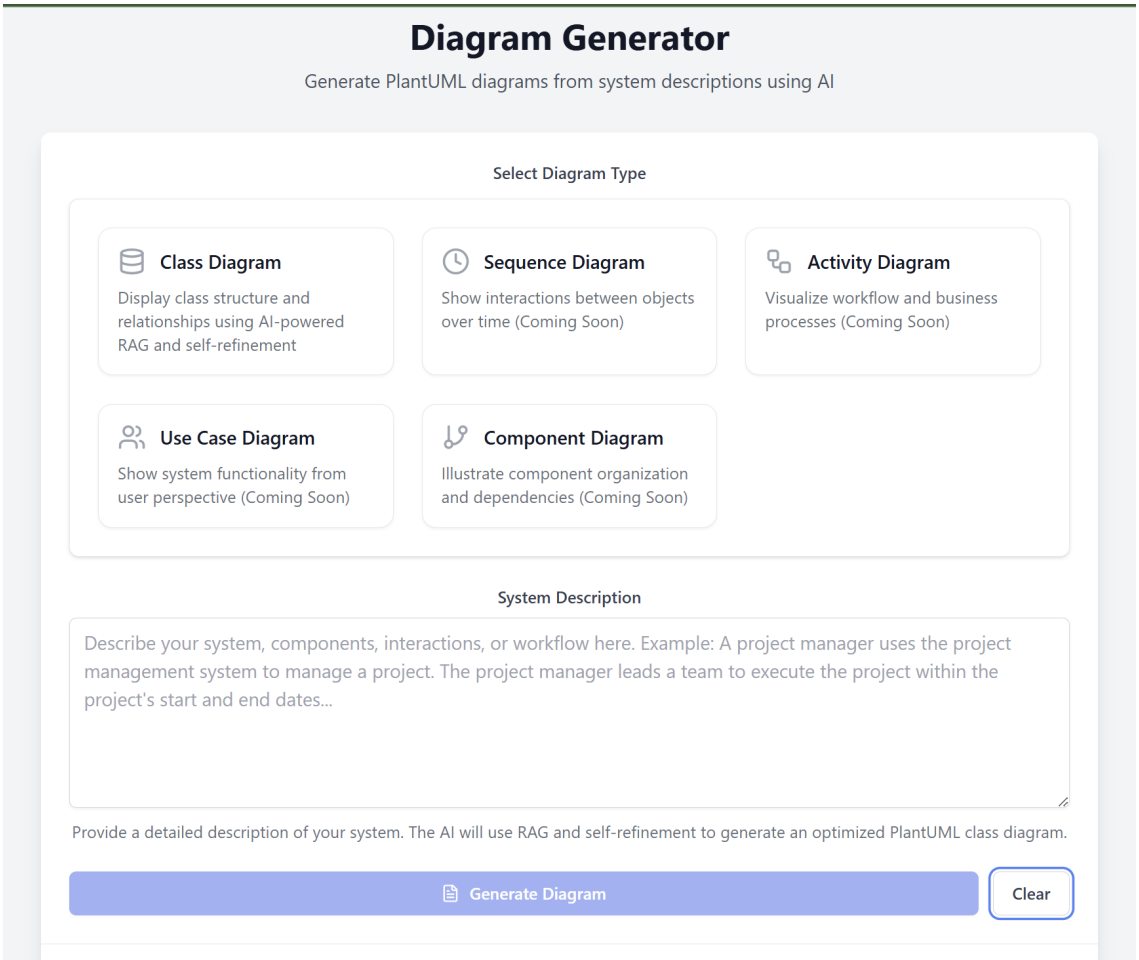


Figure 5.9: System interface for the use case

In the generated diagram (Figure 5.10), with respect to multiplicities, the application attempts to capture the requirement that "multiple medications may be prescribed during each visit." However, representing this as a direct Visit–Medication association is imprecise. The ground-truth diagram is stronger here: it introduces a Prescription association object that links a Visit to a Medication and carries visit-specific data (e.g., dosage, refills), as it can be seen in Figure 5.11. This design correctly supports the multiplicities Visit 1 – 0..* Prescription and Prescription 0..* – 1 Medication, and prevents dosage from being incorrectly modeled as a property of the medication itself.

Regarding relationships, the AI-generated diagram is clearer than the ground truth: instead of just showing a line, it labels relationships with descriptive names such as "has primary care physician" or "attended by". It even models real-world scenarios more closely, such as an Appointment possibly leading to a Visit.

The AI diagram also goes beyond the specification by adding numerous operations (e.g., scheduling, cancelling appointments, getters). This can be useful in software design by adding more detail to the model.

In summary, compared with the ground truth, the principal deficiency of the AI-generated diagram is the absence of a dedicated Prescription class. Despite this, the diagram is more complete, uses class hierarchies to avoid redundancy, and includes more realistic workflows. In short, while the ground truth contains the basic concepts of the system, the AI-generated diagram represents a more functional and detailed design, making it a stronger blueprint for actual system implementation.

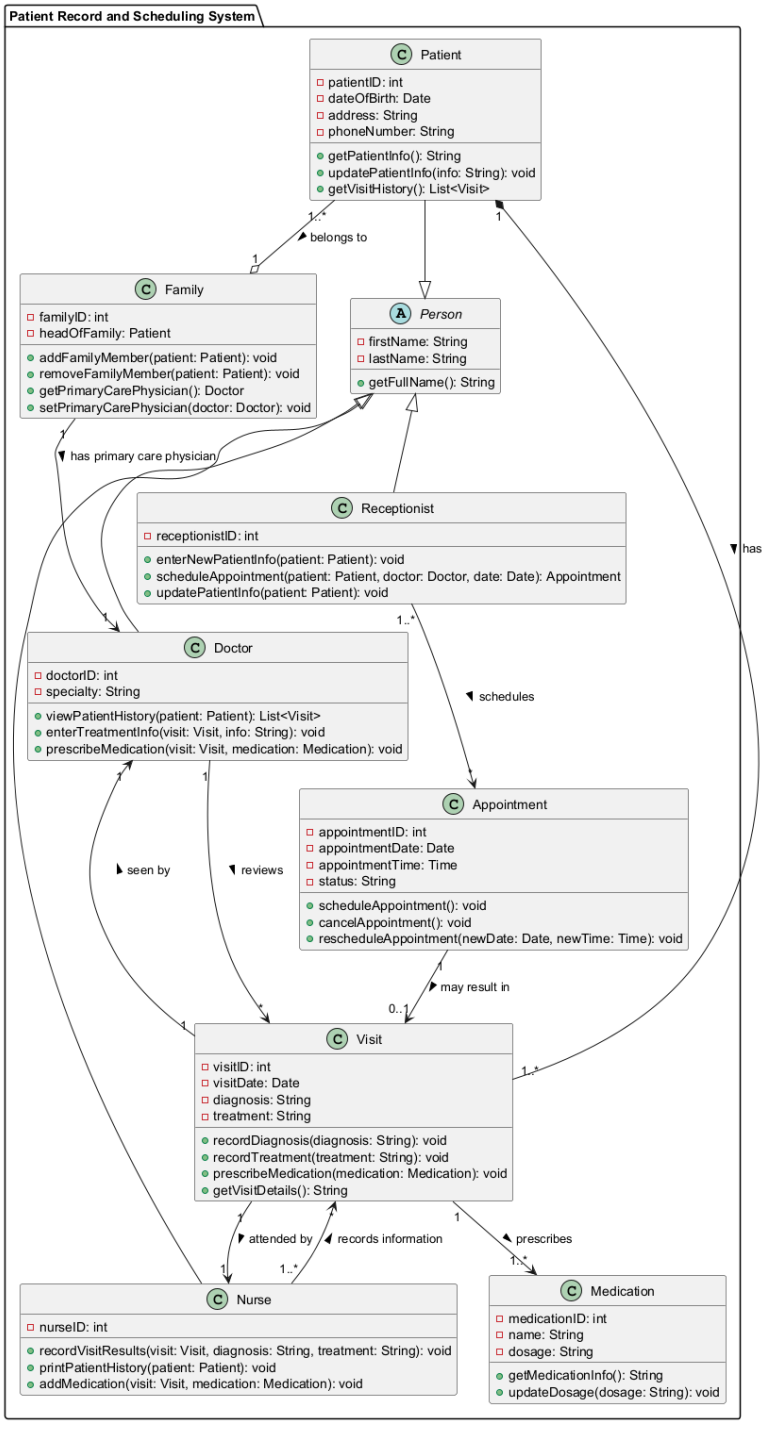


Figure 5.10: AI-generated class diagram

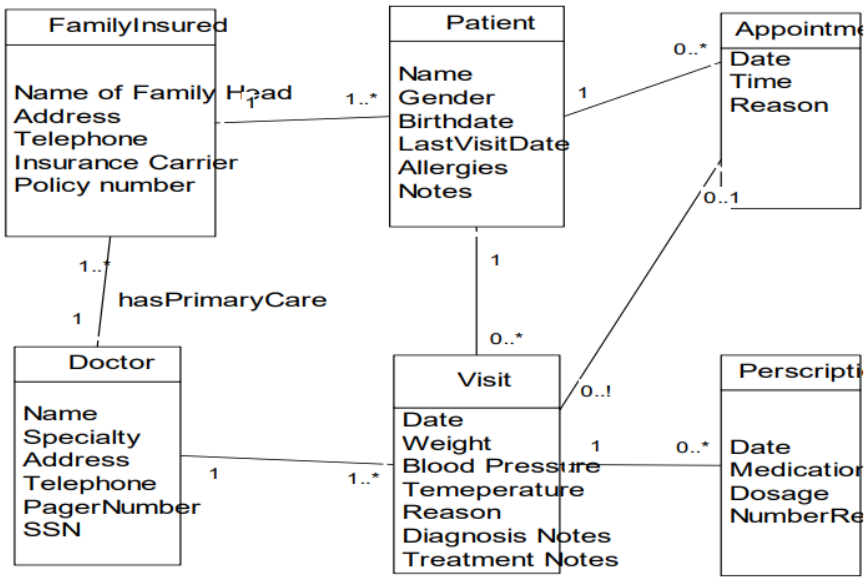


Figure 5.11: Ground truth class diagram [59]

5.7 Summary

This chapter presented the proposed approach for automating UML class diagram generation from system descriptions. It detailed the selection of the optimal prompt strategy (CoT) and LLM (Claude 3.7 Sonnet) based on empirical evaluation, the design of a lightweight and efficient RAG pipeline, and the implementation of a self-refinement loop for iterative improvement. The complete system architecture integrates generation, refinement, and rendering modules. A practical case study demonstrated the system's effectiveness, producing diagrams that were more complete and detailed than the ground truth, while highlighting areas for further refinement.

6

Evaluation

In this chapter presents the evaluation procedure of our approach, in which we used a **questionnaire** and **G-evaluation** to assess the qualities of our proposed automation pipeline to create class diagrams from system descriptions. The questionnaire used for the evaluation is detailed, explaining each question's purpose. Subsequently, the conclusions gathered from both evaluations are presented.

6.1 Experiment Purpose and Procedure

The purpose of this experiment is twofold, focusing on the evaluation of the quality and perception of generated diagrams:

- Discover if the generated diagrams can outperform the diagrams created by a human.
- Examine whether the participants can distinguish which diagrams were created with Generative AI tools and methods from those created by humans.
- Understand which diagrams, in the opinion of the participants, best represent the system description.

For this purpose, we used the same dataset as in the previous chapters (Chapter 4 and Chapter 5) to conduct the evaluation. Since this dataset [59] already contained UML Class diagrams created by the author of the paper, we used those to guide our experiment.

6.1.1 G-evaluation

We firstly evaluated the class diagrams created by our approach and the class diagrams created by a human using the same evaluation protocol described in Section 4.1. Once we had the results of each class diagram, we compared the diagrams generated by our approach with those created by humans.

6.1.1.1 Results

Analysis of the G-evaluation results presented in Figure 6.1 indicates that, overall, the proposed framework generated class diagrams with **higher overall scores on average in the 19 exercises evaluated** than those produced by humans, according to the assessment conducted by the LLM. Examining the impact of difficulty level, it is visible that human diagrams performed relatively better on diagrams describing simpler systems. However, as system descriptions became more complex, the framework consistently achieved superior results.

Furthermore, both groups exhibited a decline in overall scores as task difficulty increased, but the slope for human-generated diagrams was noticeably steeper. This trend suggests that the quality of human-generated diagrams tends to decrease more significantly as the level of difficulty rises. Notably, for systems with difficulty levels of 3 and 5, there was a marked reduction in the overall scores assigned to human-created diagrams.

In addition to overall performance, the framework performed slightly better than humans' diagrams across specific quality dimensions, including both **pragmatic** and **semantic** criteria, as evaluated by the LLM. These findings demonstrate the robustness and effectiveness of the framework, particularly for complex system descriptions and multiple quality aspects.

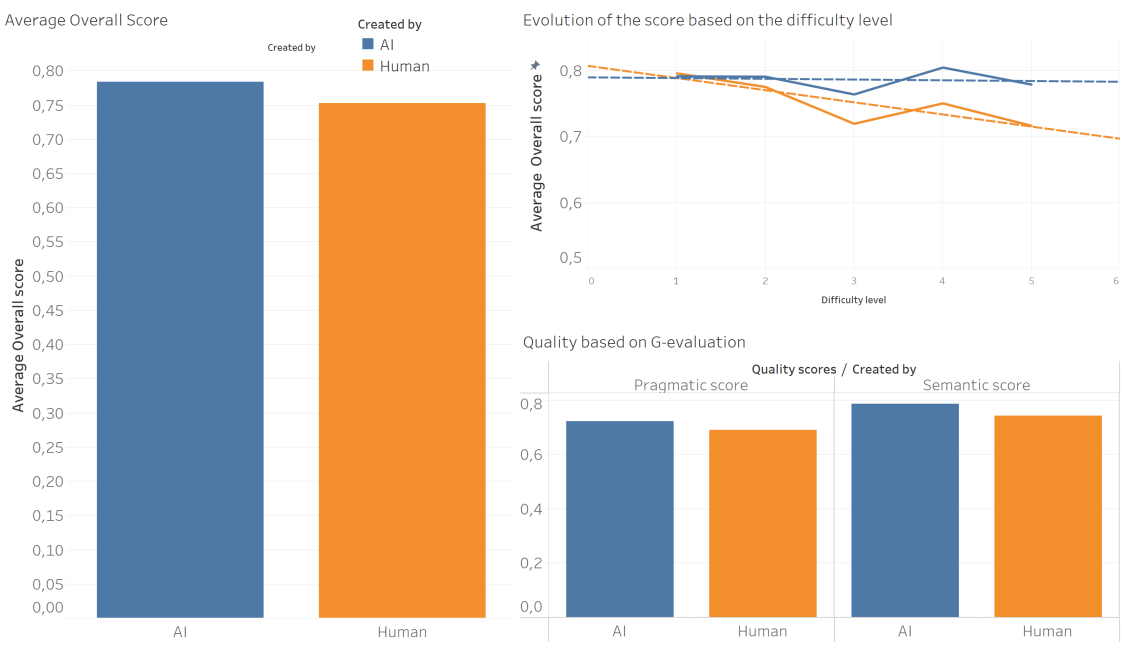


Figure 6.1: Results of G-evaluation comparing human-created diagrams and AI-generated diagrams overall and across different difficulty levels.

A detailed examination reveals important nuances in the comparative performance of AI-generated and human-created class diagrams. As illustrated in Figure 6.2, **human participants produced diagrams that surpassed those generated by the framework** in several exercises, notably in exercises 7, 11, 16, and 18. Moreover, in many tasks, the

difference in overall scores between the human and AI diagrams was minimal, indicating that the results are closely matched and making it difficult to assert a clear superiority in these cases.

Despite these instances, the overall trend observed in the G-evaluation points to **the proposed framework outperforming models created by human participants.** This finding suggests that the framework has considerable potential to assist in producing higher-quality class diagrams. Nevertheless, because the evaluation was conducted automatically by an LLM, it is important to complement these results with manual, expert assessment. Such validation is important to ensure the reliability and practical relevance of the outcomes, affirming the framework's effectiveness within real-world modeling contexts.

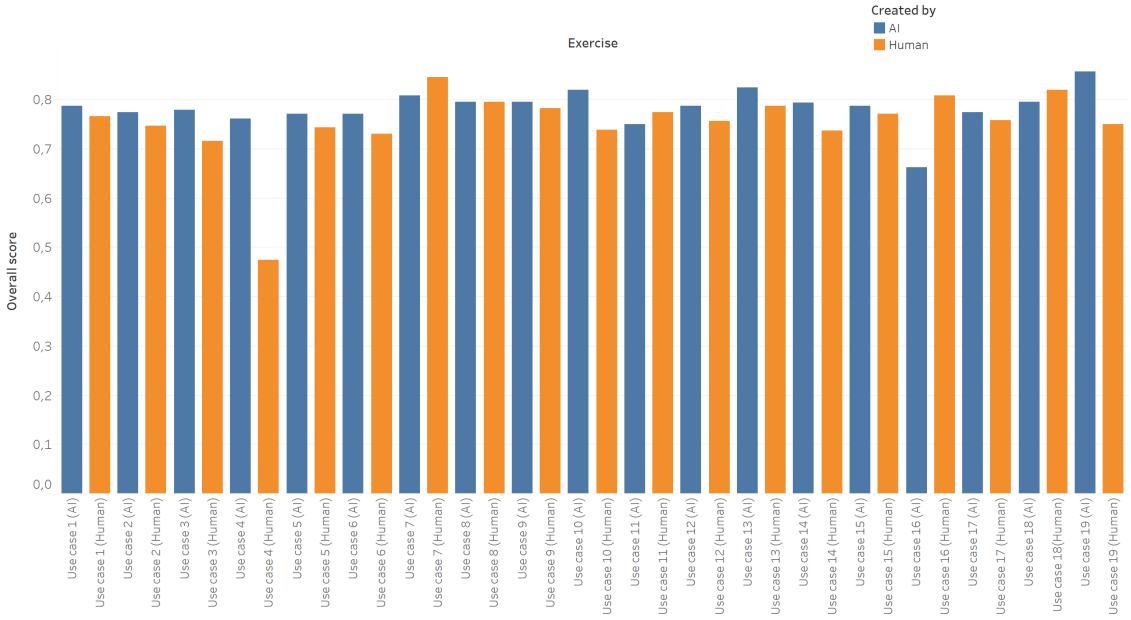


Figure 6.2: Results of G-evaluation of each of the 19 exercises.

6.1.2 Online Questionnaire

To further assess the quality of the class diagrams generated by Generative AI and by human participants, a questionnaire was developed to complement the results previously found in the G-evaluation. In this questionnaire, participants were presented with three out of the nineteen exercises from the study. For each selected exercise, participants evaluated two diagrams,one produced by Generative AI and one by a human, across three dimensions of quality: **Syntactic Quality**, **Semantic Quality Pragmatic Quality** were presented with three out of the nineteen exercises from the study.

Beyond the quality assessment, the questionnaire was designed to **investigate whether participants could accurately distinguish between diagrams generated by AI and those**

created by humans. As such, an additional question was included, asking respondents to identify which diagram was produced by Generative AI.

The decision to include only three exercises was motivated by the goal of keeping the questionnaire manageable and not be overly time-consuming for participants. The exercises selected represent a spectrum of comparative performance based on the G-evaluation results: exercise 9, where human and AI diagrams yielded similar scores; exercise 16, where the human diagram outperformed the AI; and exercise 19, where the AI diagram outperformed the human, as shown in Figure 6.2. The complete questionnaire structure is documented in Appendix C.1.

To gather expert feedback, 45 professionals with experience in UML and requirements engineering were invited to participate. Of these, 30 individuals agreed and completed the questionnaire. The survey was distributed via Google Forms, allowing easy sharing by link and facilitating accessibility and convenience for participants.

6.1.3 Section One: Explanation of research purpose and instructions for the participant

The first section of the questionnaire provides an explanation of the research purpose, as well as detailed instructions on how to evaluate the class diagrams and other relevant information. In this section, we also highlight the importance of not using Generative AI tools to complete the questionnaire, as doing so would threaten the validity of the results.

6.1.4 Section Two: Pre-Survey User Profile

The second section is a pre-survey user profile, which gathers user information to better understand our user sample. The information requested from the user is as follows:

- Current role / title
- Experience in requirements engineering
- Education background
- Industry/sector in which the participant is currently working/studying

6.1.5 Sections Three, Four and Five: Exercises for the participants to evaluate

Sections three, four, and five are structured in the same way, with only the exercises being analyzed changing. In these sections, the user is presented with examples of possible mistakes that can be present in class diagrams for each quality dimension. Next, the system description from which the class diagrams were created is shown. Then, the class diagrams, generated either by a human or by our approach, are presented. **The order in which the diagrams appear is randomized** and differs for each of the three sections.

Alongside each class diagram, the user answers three classification questions, evaluating the diagram on a scale of 1 to 5 for each quality dimension assessed. The three qualities used in this evaluation are as follows:

- **Syntactic Quality:** The user assesses whether the class diagram follows the syntactic rules of UML Class Diagrams.
- **Semantic Quality:** The user verifies whether the diagrams accurately and completely represent the intended domain by assessing two key aspects: validity (whether all elements and relationships accurately depict the domain) and completeness (whether all necessary elements and relationships are represented).
- **Pragmatic Quality:** The user analyzes how understandable the diagrams are from the stakeholders' perspective.

After both class diagrams are evaluated, the user answers two further questions. First, the user is asked to decide which class diagram was created using Generative AI tools. Second, the user is asked to indicate which diagram, in their opinion, best represents the system description presented previously.

6.1.6 Section Six: Consent and Verification of Participation Rules

In this section, we requested participants' consent and verified that the participation rules were respected. We asked whether the participant had read the entire first section, which contains detailed instructions on how to complete the questionnaire. Additionally, we confirmed whether the participant used any Generative AI tools while responding. Finally, we requested permission to use the anonymized responses for academic research purposes.

6.2 Analysis of the Results

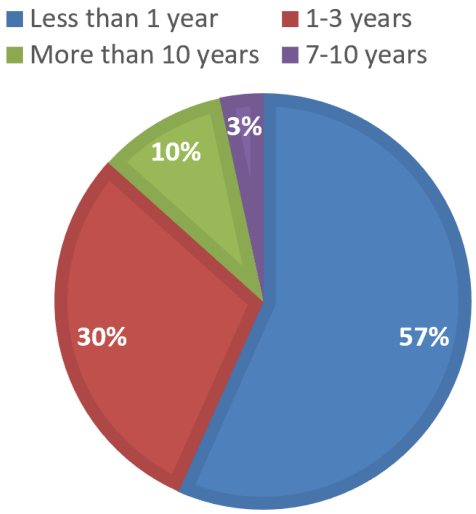
In this section, the evaluation results are presented according to the main questions of the questionnaire. First, the two diagrams from the three selected exercises are assessed based on three key qualities: **syntactic quality**, **pragmatic quality**, and **semantic quality**. Next, the analysis examines participants' ability to distinguish between diagrams created by Generative AI and those created by humans. Finally, user preferences regarding the diagrams are explored.

6.2.1 User Profile

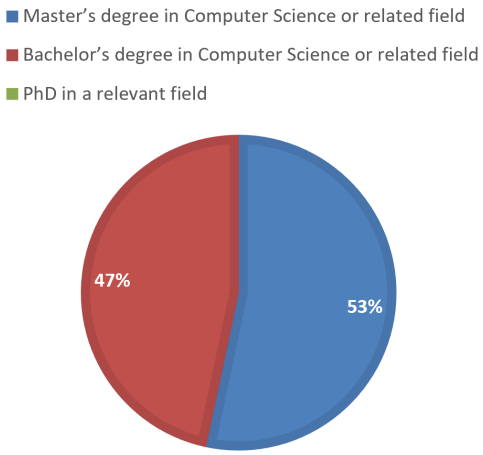
The study collected 30 responses from participants with diverse educational and professional backgrounds. Over 50% of respondents held master's degrees, with participants drawn from both students and professionals in software engineering and related fields, as it can be seen in Figure 6.3b.

Experience levels in Requirements Engineering varied considerably: while the majority of participants had less than one year of experience, the sample also included seasoned professionals with over 10 years in the field, as illustrated in Figure 6.3a. This distribution provided perspectives from both emerging practitioners and experienced specialists.

Participants represented 10 different sectors and industries, though the predominant groups were students and professionals working in software development. This diversity in professional contexts ensured a broad range of perspectives on diagram quality and usability.



(a) Participants' experience levels in Requirements Engineering.



(b) Participants' education levels.

Figure 6.3: Overview of participants' background.

6.2.2 Quality Evaluation

In terms of quality evaluation, **the class diagrams created by our Generative AI approach achieved better results in the three qualities, across the three exercises**, as can be seen in the Figures figs. 6.4 to 6.6.

- **Syntactic Quality:**
  - AI-generated class diagrams consistently scored higher than those created by humans.
  - They showed stronger adherence to UML syntax rules, such as cardinality details and appropriate naming of classes and associations.
  - *Conclusion:* The results suggest that Generative AI demonstrates relative strength in applying UML syntax correctly.
- **Semantic Quality:**

- Larger score differences were observed in the first and second exercises.
- Across all three exercises, AI-generated diagrams were rated higher than human-created ones.
- Improvements were noted in identifying relationships and representing the intended domain accurately and completely.
- *Conclusion:* The results suggest that our approach was able to improve the capacity of LLMs in identifying relationships, a problem identified in Chapter 3, and inaccurately and completely representing the intended domain. Although our approach achieved better results, it still tends to overgeneralize.

- **Pragmatic Quality:**
  - The gap between AI-generated and human diagrams was smaller compared to the other dimensions.
  - AI-generated diagrams scored higher but received the lowest overall quality scores across the three dimensions.
  - *Conclusion:* While AI performed slightly better pragmatically, diagram understandability is still the most challenging aspect.

In conclusion, our AI approach outperformed the human-created diagrams in syntactic and semantic quality, which suggests that **AI can be stronger in adhering to UML rules and covering the domain of the system**. However, the less pronounced difference in pragmatic quality suggests that **understandability is a shared concern**.

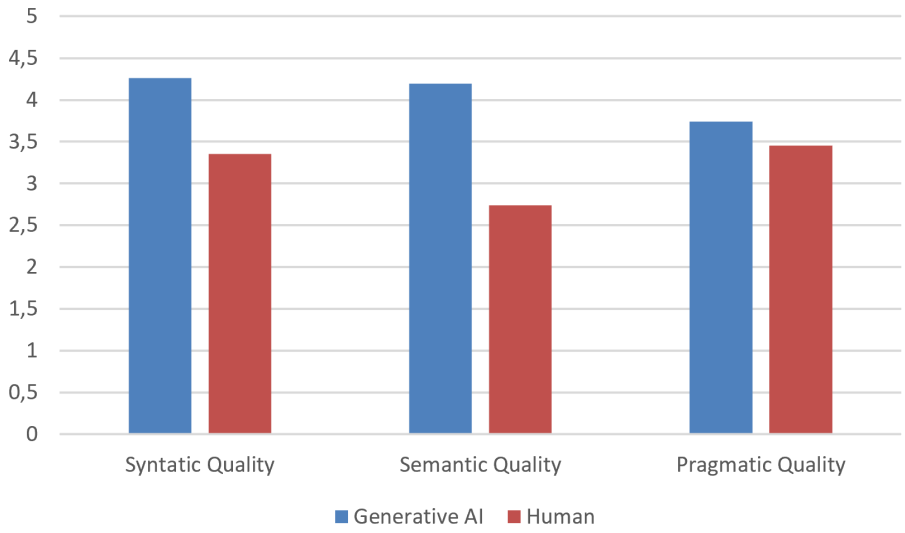


Figure 6.4: Quality scores for the first exercise.

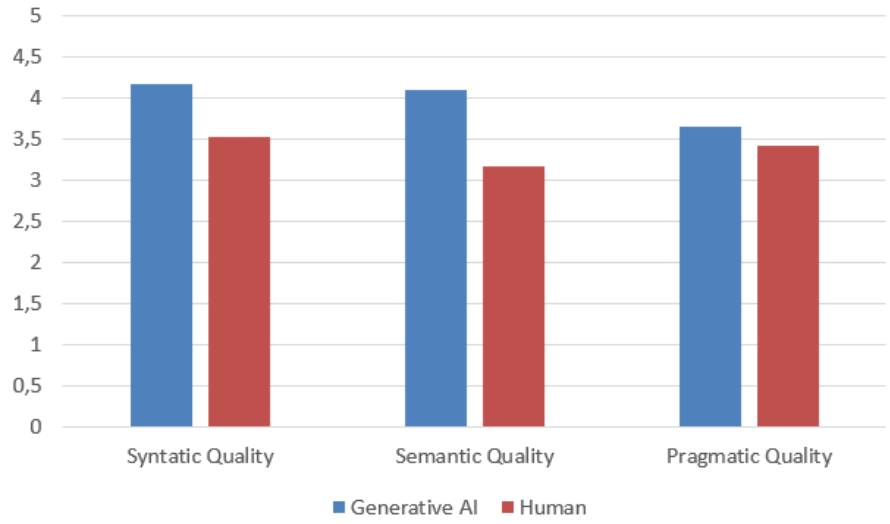


Figure 6.5: Quality scores for the second exercise.

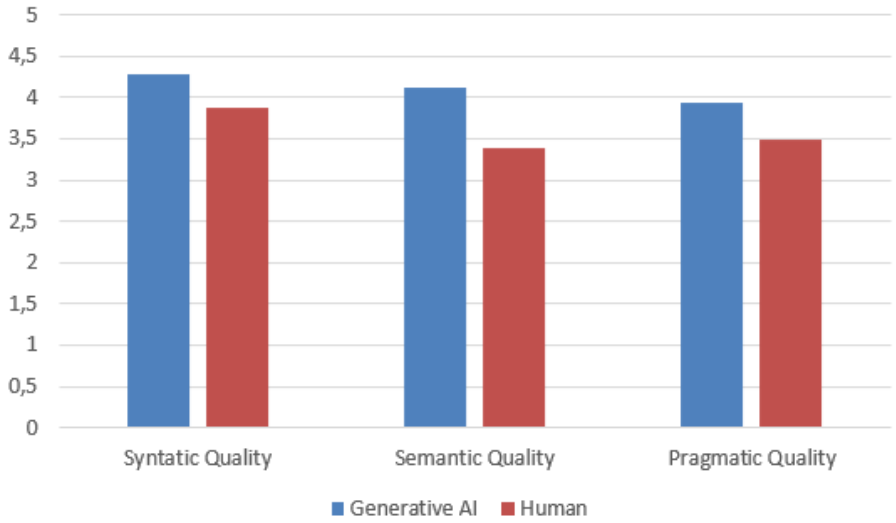


Figure 6.6: Quality scores for the third exercise.

6.2.3 Identification of the Class Diagram Generated by the AI Approach

When it comes to identifying the class diagram that was generated with AI, the results show that in **all three exercises more than 65% of the users misidentified**, as shown in Figure 6.7. The second exercise had the highest number of users incorrectly identifying the diagrams. These results suggest that **our approach is already capable of generating diagrams that most evaluators cannot distinguish from human work**, showing that the class diagrams generated by our approach are visually and structurally close to those created by humans.

However, the results also raise caution: if **users cannot reliably recognize AI-generated diagrams, there could be risks if such diagrams contain subtle errors that go unnoticed**, especially if validation relies too much on surface features.

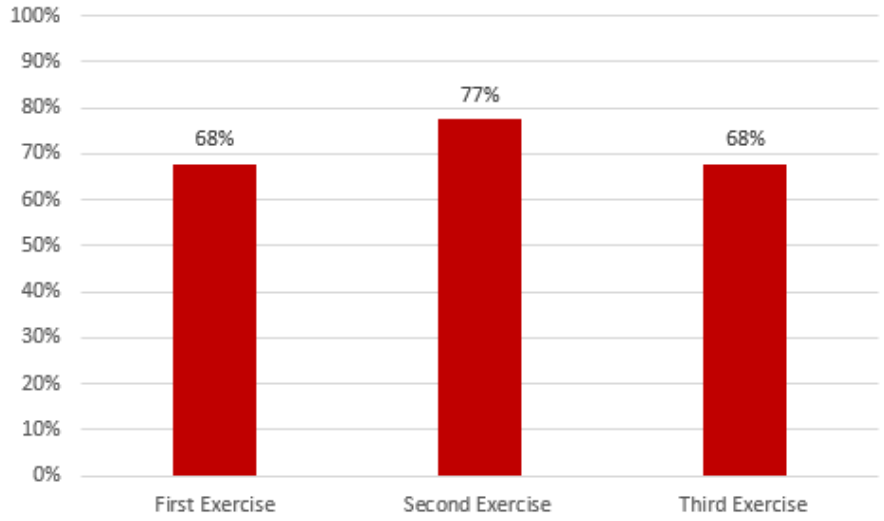


Figure 6.7: Participants' ability to identify AI-generated diagrams.

6.2.4 Preferred Diagram to Represent the System Description

Participants showed a **clear preference for AI-generated class diagrams** over human-created ones across all exercises, as can be seen in Figure 6.8. These results indicate that our approach produces models that users find clearer, more useful, and more representative of system descriptions than traditional methods.

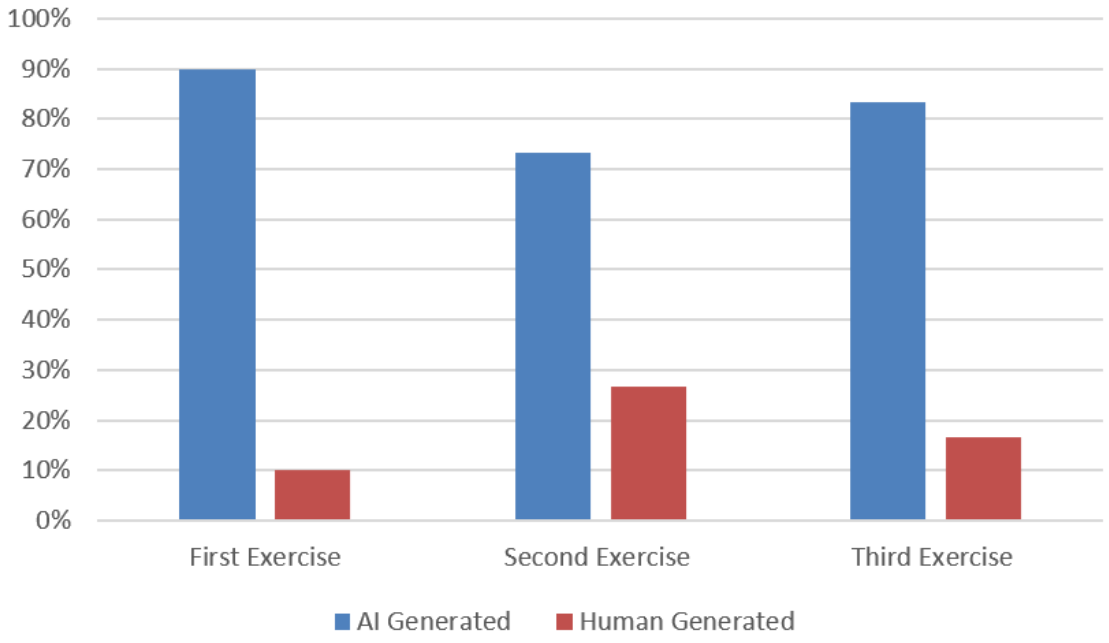


Figure 6.8: Participants' preferred diagram types.

6.3 Discussion

The results analysis reveals several interesting conclusions. **Diagrams created with our approach achieved higher classification in both evaluations.**

In the first evaluation, we saw that on average our approach delivered better overall results. A notable trend emerged: while both groups showed score declines with increasing difficulty, the human-generated diagrams declined more rapidly, particularly for systems rated difficulty 3 and 5. This may be attributed to AI models' improved pattern recognition capabilities for UML and the fact that AI errors are often systematic, not deteriorating as quickly with complexity.

The second evaluation, conducted by software engineering professionals and students with knowledge and experience in requirements engineering, **reaffirmed the G-evaluation results**. The diagrams generated by our approach again achieved better results across all three exercises.

Examining the quality scores, we found that syntactic and semantic quality showed significant differences compared to human diagrams, indicating that **Generative AI demonstrates greater adherence to UML rules and conventions**. Previous research highlighted LLMs' difficulties in identifying relationships, but our approach overcame these challenges, achieving good scores in semantic quality. For pragmatic quality, the difference was less pronounced and represented the lowest score of the three qualities, suggesting that **Generative AI still struggles with creating class diagrams that stakeholders can easily comprehend**. This highlights one of the main challenges, **AI-generated diagrams tend to be overgeneralized and verbose**.

Importantly, more than **65% of participants misidentified** which diagrams were AI-generated, indicating that the models produced by our framework are already visually and structurally comparable to human work. Additionally, participants **consistently preferred AI-generated diagrams**, considering them clearer and more representative of the system descriptions.

This demonstrates that approaches using Generative AI can already create professional-looking artifacts. Furthermore, participants' preference for AI-generated class diagrams suggests that our approach can improve stakeholder engagement by delivering more complete and useful diagrams.

However, the fact that most participants were not able to distinguish AI-generated diagrams presents a risk in identifying subtle errors, due to reliance on surface features alone.

Overall, these results show that **Generative AI methods and tools have advanced to a point where their diagrams are not only competitive but, in many cases, preferred by the user**.

6.4 Threats to Validity

External Validity Since most participants in the experiment were students or individuals with less than one year of experience, the findings may not fully represent the views of professionals. Furthermore, the limited number of exercises and participants constrains the generalizability of the results.

Construct Validity The LLM model used in the G-evaluation may have inherent biases that could influence the assessment outcomes. As no dedicated study was conducted to analyze the quality of the model used as a judge, the results may not fully reflect the actual quality of the diagrams.

Another limitation is that the experiment focused only on UML Class diagrams, meaning that the results may not be applicable to other types of diagrams or modeling languages. Similarly, the system descriptions used may not fully represent the diversity and complexity of real-world requirements.

Internal Validity The psychological factors that may have influenced participants' evaluations, such as fatigue or bias towards AI-generated content, were not controlled for in the study, which may have affected the results.

6.5 Summary

This chapter presented a thorough evaluation of our AI-driven approach to generating class diagrams from system descriptions. The evaluation employed both questionnaires and G-evaluation techniques to assess diagram quality along three dimensions: syntactic, semantic, and pragmatic quality.

The experiment involved 30 professionals and students with expertise in UML and requirements engineering who evaluated class diagrams created by both humans and our AI approach.

Key findings include

- Our AI-generated diagrams consistently outperformed human-created diagrams across all three quality dimensions in all exercises.
- AI approach showed particular strength in syntactic quality, demonstrating superior adherence to UML rules.
- In terms of semantic quality, our approach showed marked improvement over previous LLM limitations in identifying relationships and representing domains accurately.
- While still outperforming human diagrams, pragmatic quality showed the smallest gap and the lowest score out of the three qualities, indicating that diagram understandability remains a challenge.

- Over 65% of participants could not accurately identify which diagrams
- Participants showed a clear preference for AI-generated diagrams across all exercises.

The results demonstrate that our Generative AI approach has advanced to a point where it not only produces professional-quality diagrams that are indistinguishable from human work but are often preferred by users. However, this raises concerns about relying solely on visual assessment for validating diagrams, as subtle errors might go undetected.

Overall, this evaluation confirms that our approach effectively addresses previous limitations of LLMs in creating UML class diagrams and offers a valuable tool for software engineering professionals.

7

Conclusions and Future Work

The automatic creation of UML Class diagrams presents a significant opportunity to improve and accelerate software development for enterprises by reducing the time needed for diagram creation during software modeling. This thesis introduces an automatic pipeline that leverages Generative AI to transform natural language system descriptions into UML Class diagrams. We detail the various experiments and findings required to design the pipeline, exploring LLM choices and prompting techniques, and using these resources for experiments to assess their feasibility in our research.

The proposed pipeline combines CoT prompting with a RAG system that enhances the initial prompt with relevant chunks of traditional UML documentation used when creating UML diagrams. We then apply a self-refinement technique, first proposed by Madaan et al. [66], to improve the initial generated class diagram. Results are presented as images, but users also have access to the PlantUML code, allowing them to adapt diagrams to their preferences. A proof of concept demonstrates how the pipeline transforms system descriptions of a Patient Record and Scheduling System into representative UML Class diagrams, showcasing both the strengths and limitations of Generative AI transformation.

We also provide an evaluation of the Proof of Concept diagram, presenting a comparative analysis between diagrams created by Daniel De Bari et al. present on the dataset used and diagrams created using our approach [59]. Analysis of the evaluation results reveals that diagrams generated with our pipeline achieve better results on average in both manual and automatic evaluations. However, our approach still faces challenges in creating diagrams that stakeholders can easily comprehend, highlighting the importance of human revision of the final output. Finally, we present the challenges associated with this research, demonstrating how emerging technologies can provide significant benefits while introducing unforeseen complexities and unresolved issues.

7.1 Contributions

In this thesis, we have made some contributions, which are summarized as follows:

1. A systematic mapping study of different techniques used to automate design modeling from requirements, providing a structured overview of current approaches and identifying gaps in existing research.

2. A structured prompt template to translate system descriptions into UML class diagrams and other UML diagrams. This template addresses the ambiguity, vagueness, and incompleteness often found in real-world requirements, resulting in more reliable outputs from generative models.

3. A simple Retrieval-Augmented Generation (RAG) system to enhance the prompts provided to Generative AI systems in the context of UML diagram generation.

4. A self-refinement cycle for improving UML diagrams and mitigating inconsistent output quality.

5. A novel automatic evaluation framework for UML class diagrams based on three quality dimensions: syntactic, semantic, and pragmatic quality.

6. Findings that can inform and contribute to ongoing debates on the role of AI in software modeling.

7.2 Limitations

Despite the contributions of this research, several limitations should be acknowledged. In the systematic mapping study (SMS), there is the risk of not including all relevant studies. Although considerable effort was made to design comprehensive search strings by incorporating synonyms and alternative terms related to the research questions and the use of different digital libraries, some relevant works may have been missed if they did not use the specified keywords. Similarly, as the selection and data extraction were conducted by a single researcher, there is a residual risk of subjective bias, even though explicit inclusion and exclusion criteria were defined to mitigate this. Another limitation of the SMS comes from the choice of sources: only IEEE Xplore and ACM Digital Library were consulted, which, while being well-established repositories in the software engineering field, excluded other venues such as Springer, ArXiv, or non-English publications. In addition, the time window considered excluded research published before 2019, which may have limited the inclusion of more recent contributions.

The evaluation of the proposed approach is subject to further threats. Most participants were students or professionals with less than one year of experience, meaning the findings may not fully reflect the views of professional experts. Moreover, the limited number of participants and exercises reduces the generalizability of the results. The evaluation also focused exclusively on UML class diagrams and on a specific set of system descriptions, which may not adequately represent the diversity and complexity of real-world modeling scenarios. Another limitation concerns the use of an LLM as the judge in the G-evaluation,

as no dedicated validation study of this model was performed, its inherent biases may have influenced the outcomes, raising questions about whether the scores fully reflect the actual quality of the diagrams. In addition, no measures were taken to control psychological factors of the participants, such as fatigue or predisposition toward AI-generated content, which could have affected the results. Similarly, participants' limited availability may also have influenced the attention devoted to the questionnaire. The human baseline used for comparison was also limited, as it was based on diagrams produced by a single individual rather than a broader pool of practitioners with varying expertise.

Finally, practical constraints introduced further limitations. The proposed pipeline relies on commercial LLMs APIs, which are subject to cost and availability restrictions that may limit reproducibility and accessibility. External factors, such as hardware constraints, limited the experimentation with different Generative AI models and techniques, reducing the scope of comparative analysis.

Despite these limitations, the adoption of a structured methodology, comprehensive documentation of procedures, and the use of multiple complementary evaluation methods contribute to the overall credibility of the findings. Transparency regarding these constraints enables readers to better understand both the contributions and boundaries of this research.

7.3 Future Work

This research represents an initial step toward the automation of UML diagram modeling through Generative AI applied to natural language requirements, yet much remains to be explored.

A first avenue for future work involves **extending the proposed pipeline beyond class diagrams to other UML notations**, such as sequence diagrams, activity diagrams, and component diagrams. This would provide valuable insights into whether AI can adequately capture not only structural but also dynamic aspects of systems.

Another important direction is to evaluate the pipeline with **real-world requirements that are often ambiguous, incomplete, or conflicting**. The current evaluation relies primarily on well-structured, academically curated requirements specifications. Such an extension would allow for a more rigorous assessment of its robustness in practical and complex contexts.

In addition, the relatively **lower pragmatic scores observed in the evaluation suggest the need for further investigation**. Conducting qualitative interviews with domain experts could help to identify the underlying causes of these difficulties and inform strategies to improve diagram understandability.

Finally, future research should include field studies to examine the **practical impact of automated frameworks such as the one presented here**. These studies could shed light on whether AI-assisted modeling translates into measurable gains in productivity and error reduction in real development environments. While there is strong potential

for efficiency improvements, it is anticipated that human oversight will remain essential to ensure that all requirements are accurately represented, regardless of technological advances.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on pp. i, ii).

[2] K. Chen et al. "Automated domain modeling with large language models: a comparative study". In: *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Västerås, Sweden: IEEE, 2023, pp. 162–172. DOI: 10.1109/MODELS58315.2023.00037 (cit. on pp. 1, 24, 25, 85).

[3] M. Jahan et al. "Automated derivation of uml sequence diagrams from user stories: unleashing the power of generative ai vs. a rule-based approach". In: *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS '24)*. New York, NY, USA: Association for Computing Machinery, 2024, pp. 138–148. DOI: 10.1145/3640310.3674081. URL: https://doi.org/10.1145/3640310.3674081 (cit. on pp. 1, 20–25, 84).

[4] Z. Babaalla et al. "Towards an automatic extracting uml class diagram from system's textual specification". In: *Proceedings of the 7th International Conference on Networking, Intelligent Systems and Security (NISS '24)*. New York, NY, USA: Association for Computing Machinery, 2024, Article 36, 1–5. DOI: 10.1145/3659677.3659742. URL: https://doi.org/10.1145/3659677.3659742 (cit. on pp. 1, 20, 22–25, 84).

[5] J. W. P. Miranda et al. "Towards an in-context llm-based approach for automating the definition of model views". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE '24)*. New York, NY, USA: Association for Computing Machinery, 2024, pp. 29–42. DOI: 10.1145/3687997.3695650. URL: https://doi.org/10.1145/3687997.3695650 (cit. on p. 1).

[6] A. Ferrari, S. Abualhaija, and C. Arora. "Model generation with llms: from requirements to uml sequence diagrams". In: *2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW)*. Reykjavik, Iceland: IEEE, 2024, pp. 291–300. DOI: 10.1109/REW61692.2024.00044 (cit. on pp. 1, 2, 20, 22–24, 84).

[7] K. Ruan, X. Chen, and Z. Jin. "Requirements modeling aided by chatgpt: an experience in embedded systems". In: *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*. Hannover, Germany: IEEE, 2023, pp. 170–177. DOI: 10.1109/REW57809.2023.00035 (cit. on pp. 1, 20, 22–25, 45, 85).

[8] M. A. Omar. "Measurement of chatgpt performance in mapping natural language specification into an entity relationship diagram". In: *2023 IEEE 11th International Conference on Systems and Control (ICSC)*. Sousse, Tunisia: IEEE, 2023, pp. 530–535. DOI: 10.1109/ICSC58660.2023.10449869 (cit. on pp. 2, 20, 22–24, 85).

[9] B. Wang et al. "How llms aid in uml modeling: an exploratory study with novice analysts". In: *2024 IEEE International Conference on Software Services Engineering (SSE)*. Shenzhen, China: IEEE, 2024, pp. 249–257. DOI: 10.1109/SSE62657.2024.00046 (cit. on pp. 2, 22–24, 84).

[10] H. Gomaa. *Software modeling and design: uml, use cases, patterns, and software architectures*. Cambridge, UK: Cambridge University Press, 2011 (cit. on pp. 5–8).

[11] G. Booch, J. Rumbaugh, and I. Jacobson. *The unified modeling language reference manual*. 1st. Addison-Wesley, 1999. ISBN: 978-0201309983 (cit. on p. 5).

[12] M. Sergievskiy and K. Kirpichnikova. "Optimizing uml class diagrams". In: *ITM Web of Conferences*. Vol. 18. EDP Sciences, 2018, p. 03003 (cit. on p. 5).

[13] A. Hafeez et al. "Importance and impact of class diagram in software development". In: *Indian Journal of Science and Technology* (2025). [Accessed: Jan. 23, 2025]. DOI: 10.17485/ijst/2019/v12i25/145739 (cit. on p. 6).

[14] M. Zapata-Barra et al. "Towards obtaining uml class diagrams from secure business processes using security patterns". In: *Journal of Universal Computer Science* 24.10 (2018), pp. 1472–1492 (cit. on p. 6).

[15] Microsoft Corporation. *Microsoft visio*. Diagramming and vector graphics application. 2025. URL: https://www.microsoft.com/en-us/microsoft-365/visio/flowchart-software (cit. on p. 8).

[16] MKLabs Co., Ltd. *Staruml*. UML modeling tool. 2025. URL: http://staruml.io (cit. on p. 8).

[17] Arnaud Roques. *Plantuml*. Open-source tool for UML diagrams. 2025. URL: http://plantuml.com (cit. on pp. 8, 20).

[18] Lucid Software Inc. *Lucidchart*. Online diagramming application. 2025. URL: https://www.lucidchart.com (cit. on p. 8).

[19] T. L. Scao et al. *What Language Model to Train if You Have One Million GPU Hours?* 2022. arXiv: 2210.15424 [cs.CL]. URL: https://arxiv.org/abs/2210.15424 (cit. on p. 8).

[20] J. Clusmann et al. "The future landscape of large language models in medicine". In: *Communications Medicine* 3.1 (2023-10-10), p. 141. ISSN: 2730-664X. DOI: 10.1038/s43856-023-00370-1. URL: https://doi.org/10.1038/s43856-023-00370-1 (cit. on p. 8).

[21] U. P. Liyanage and N. D. Ranaweera. "Ethical considerations and potential risks in the deployment of large language models in diverse societal contexts". In: *Journal of Computational Social Dynamics* 8.11 (2023), pp. 15–25 (cit. on p. 8).

[22] A. Vaswani et al. "Attention is all you need". In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017 (cit. on p. 8).

[23] T. B. Brown et al. *Language models are few-shot learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165 (cit. on p. 9).

[24] Z. Jiang et al. "How can we know what language models know?" In: *Transactions of the Association for Computational Linguistics* 8 (2020), pp. 423–438 (cit. on p. 9).

[25] N. Thakur et al. "Beir: a heterogenous benchmark for zero-shot evaluation of information retrieval models". In: *arXiv preprint arXiv:2104.08663* (2021) (cit. on p. 10).

[26] Y. Zhang et al. "Siren's song in the ai ocean: a survey on hallucination in large language models". In: *arXiv preprint arXiv:2309.01219* (2023) (cit. on p. 10).

[27] Y. Gao et al. "Retrieval-augmented generation for large language models: a survey". In: *arXiv preprint arXiv:2312.10997* (2023) (cit. on p. 10).

[28] P. Lewis et al. "Retrieval-augmented generation for knowledge-intensive nlp tasks". In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. 2020, pp. 9459–9474 (cit. on p. 10).

[29] S. Robertson and H. Zaragoza. "The probabilistic relevance framework: bm25 and beyond". In: *Foundations and Trends in Information Retrieval* 3.4 (2009), pp. 333–389 (cit. on p. 10).

[30] V. Karpukhin et al. "Dense passage retrieval for open-domain question answering". In: *arXiv preprint arXiv:2004.04906* (2020) (cit. on p. 10).

[31] S. Gupta, R. Ranjan, and S. N. Singh. *A comprehensive survey of retrieval-augmented generation (rag): evolution, current landscape and future directions*. 2024. arXiv: 2410.12837 [cs.CL]. URL: https://arxiv.org/abs/2410.12837 (cit. on p. 11).

[32] P. Lewis et al. *Retrieval-augmented generation for knowledge-intensive nlp tasks*. 2021. arXiv: 2005.11401 [cs.CL]. URL: https://arxiv.org/abs/2005.11401 (cit. on p. 10).

[33] P. Lewis et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Advances in neural information processing systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf (cit. on p. 11).

[34] Y. Gao et al. *Modular rag: transforming rag systems into lego-like reconfigurable frameworks*. 2024. arXiv: 2407.21059 [cs.CL]. URL: https://arxiv.org/abs/2407.21059 (cit. on p. 11).

[35] J. K. *Types of rag: an overview*. Accessed: February 7, 2025. 2024. URL: https://blog.jayanthk.in/types-of-rag-an-overview-0e2b3ed71b82 (cit. on p. 12).

[36] S. K. et al. *Guidelines for performing systematic literature reviews in software engineering*. Tech. rep. Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007 (cit. on p. 14).

[37] K. Petersen et al. "Systematic mapping studies in software engineering". In: *12th International Conference on Evaluation and Assessment in Software Engineering*. Vol. 17. 2008, p. 1 (cit. on pp. 14, 17).

[38] B. A. Kitchenham. "Systematic review in software engineering: where we are and where we should be going". In: *Proceedings of the 2nd International Workshop on Evidential Assessment of Software Technologies*. EAST '12. Lund, Sweden: Association for Computing Machinery, 2012, 1–2. ISBN: 9781450315098. DOI: 10.1145/2372233.2372235. URL: https://doi.org/10.1145/2372233.2372235 (cit. on p. 14).

[39] L. Zhao et al. "Natural language processing for requirements engineering: a systematic mapping study". In: *ACM Comput. Surv.* 54.3 (2022), Article 55, 41 pages. DOI: 10 (cit. on p. 15).

[40] IEEE Xplore. *Ieee xplore digital library*. Accessed 01/18/2025. 2025. URL: https://ieeexplore.ieee.org/Xplore/home.jsp (cit. on p. 18).

[41] Acm digital library. *Acm digital library*. Accessed 01/18/2025. 2025. URL: https://dl.acm.org/ (cit. on p. 18).

[42] D. D. Bari et al. "Evaluating large language models in exercises of uml class diagram modeling". In: *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '24)*. New York, NY, USA: Association for Computing Machinery, 2024, pp. 393–399. DOI: 10.1145/3674805.3690741. URL: https://doi.org/10.1145/3674805.3690741 (cit. on pp. 20–24, 84).

[43] T. Eisenreich, S. Speth, and S. Wagner. "From requirements to architecture: an ai-based journey to semi-automatically generate software architectures". In: *Proceedings of the 1st International Workshop on Designing Software (Designing '24)*. New York, NY, USA: Association for Computing Machinery, 2024, pp. 52–55. DOI: 10.11

45/3643660.3643942. URL: https://doi.org/10.1145/3643660.3643942 (cit. on pp. 20–25, 84).

[44] Y. Li et al. "Llm-based class diagram derivation from user stories with chain-of-thought promptings". In: *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. Osaka, Japan: IEEE, 2024, pp. 45–50. DOI: 10.1109/COMPSAC61105.2024.00017 (cit. on pp. 20–24, 85).

[45] Z. Wang et al. "The application of llms in the analysis and modeling of software requirements". In: *2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. Cambridge, United Kingdom: IEEE, 2024, pp. 1143–1153. DOI: 10.1109/QRS-C63300.2024.00151 (cit. on pp. 20, 22–24, 84).

[46] P. Ardimento, M. L. Bernardi, and M. Cimitile. "Teaching uml using a rag-based llm". In: *2024 International Joint Conference on Neural Networks (IJCNN)*. Yokohama, Japan: IEEE, 2024, pp. 1–8. DOI: 10.1109/IJCNN60899.2024.10651492 (cit. on pp. 21–25, 85).

[47] M. B. Chaaben, L. Burgueño, and H. Sahraoui. "Towards using few-shot prompt learning for automating model completion". In: *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. Melbourne, Australia: IEEE/ACM, 2023, pp. 7–12. DOI: 10.1109/ICSE-NIER58687.2023.00008 (cit. on pp. 21–25, 45, 85).

[48] A. Ahmad et al. "Towards human-bot collaborative software architecting with chatgpt". In: *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering (EASE '23)*. New York, NY, USA: Association for Computing Machinery, 2023, pp. 279–285. DOI: 10.1145/3593434.3593468. URL: https://doi.org/10.1145/3593434.3593468 (cit. on pp. 21, 23–25, 84).

[49] L. Naimi et al. "A new approach for automatic test case generation from use case diagram using LLMs and prompt engineering". In: *2024 International Conference on Circuit, Systems and Communication (ICCSC)*. 2024, pp. 1–5. DOI: 10.1109/ICCSC62074.2024.10616548 (cit. on pp. 23, 85).

[50] OpenAI. *Openai models documentation*. Accessed: January 18, 2025. 2025. URL: https://platform.openai.com/docs/models (cit. on p. 24).

[51] Llama AI. *Llama ai platform*. Accessed: January 18, 2025. 2025. URL: https://www.llama.com/ (cit. on p. 24).

[52] T. GLM et al. *Chatglm: a family of large language models from glm-130b to glm-4 all tools*. 2024. arXiv: 2406.12793 [cs.CL]. URL: https://arxiv.org/abs/2406.12793 (cit. on p. 24).

[53] L. Zhao et al. "Natural language processing for requirements engineering: a systematic mapping study". In: *ACM Computing Surveys* 54.3 (2021), 55:1–55:41. DOI: 10.1145/3444689 (cit. on pp. 26, 28).

[54] R. Dhar, K. Vaidhyanathan, and V. Varma. "Can llms generate architectural design decisions? an exploratory empirical study". In: *Proceedings of the IEEE 21st International Conference on Software Architecture (ICSA)*. IEEE, 2024, pp. 79–89. DOI: 10.1109/ICSA59870.2024.00016 (cit. on pp. 27, 28).

[55] B. Chen, F. Yi, and D. Varró. "Prompting or fine-tuning? a comparative study of large language models for taxonomy construction". In: *Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 2023, pp. 588–597. DOI: 10.1109/MODELS-C59198.2023.00097 (cit. on pp. 27, 28).

[56] S. J. Ali, V. Naganathan, and D. Bork. "Establishing traceability between natural language requirements and software artifacts by combining rag and llms". In: *Proceedings of the International Conference on Conceptual Modeling (ER)*. Springer, 2024 (cit. on p. 27).

[57] C. Arora, T. Herda, and V. Homm. "Generating test scenarios from nl requirements using retrieval-augmented llms: an industrial study". In: *Proceedings of the IEEE 32nd International Requirements Engineering Conference (RE)*. IEEE, 2024. DOI: 10.1109/RE59067.2024.00031 (cit. on p. 28).

[58] S. J. Ali, I. Reinhartz-Berger, and D. Bork. "How are llms used for conceptual modeling? an exploratory study on interaction behavior and user perception". In: *Proceedings of the International Conference on Conceptual Modeling (ER)*. Springer, 2024 (cit. on pp. 28, 29).

[59] D. D. Bari. "Thesis Database". In: (2024-02). DOI: 10.6084/m9.figshare.25316440.v2. URL: https://figshare.com/articles/dataset/Thesis_Database/25316440 (cit. on pp. 30, 34, 56, 60, 61, 73).

[60] Y. Liu et al. *G-eval: nlg evaluation using gpt-4 with better human alignment*. 2023. arXiv: 2303.16634 [cs.CL]. URL: https://arxiv.org/abs/2303.16634 (cit. on p. 32).

[61] C.-Y. Lin. "ROUGE: A Package for Automatic Evaluation of Summaries". In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, 2004-07, pp. 74–81. URL: https://aclanthology.org/W04-1013/ (cit. on p. 32).

[62] R. Martin and S. Johnson. *Introducing notebooklm*. Google Blog. 2023-07. URL: https://blog.google/technology/ai/notebooklm-google-ai/ (cit. on p. 33).

[63] R. Miles and K. Hamilton. *Learning uml 2.0*. Sebastopol, CA: O'Reilly Media, 2006. ISBN: 9780596009823 (cit. on p. 34).

[64] Object Management Group. *Unified modeling language (uml), version 2.5.1*. Tech. rep. Object Management Group, 2017. URL: https://www.omg.org/spec/UML/2.5.1 (cit. on p. 34).

[65] PlantUML. *Plantuml language reference guide*. Accessed 07/25/2025. n.d. URL: https://plantuml.com/guide (cit. on p. 34).

[66] A. Madaan et al. "Self-Refine: Iterative Refinement with Self-Feedback". In: *Advances in neural information processing systems*. Ed. by A. Oh et al. Vol. 36. Curran Associates, Inc., 2023, pp. 46534–46594. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf (cit. on pp. 39, 52, 73).

[67] J. Wei et al. *Chain-of-thought prompting elicits reasoning in large language models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903 (cit. on p. 53).

A

Papers selected during the systematic mapping study

Authors	Title	Reference
Daniele De Bari, Giacomo Garaccione, Riccardo Coppola, Marco Torchiano, Luca Ardito	Evaluating Large Language Models in Exercises of UML Class Diagram Modeling	[42]
Zakaria Babaalla, Abdeslam Jakimi, Mohamed Oualla, Rachid Saadane, Abdellah Chehri	Towards an Automatic Extracting UML Class Diagram from System's Textual Specification	[4]
Aakash Ahmad, Muhammad Waseem, Peng Liang, Mahdi Fahmideh, Mst Shamima Aktar, Tommi Mikkonen	Towards Human-Bot Collaborative Software Architecting with ChatGPT	[48]
Munima Jahan, Mohammad Mahdi Hassan, Reza Golpayegani, Golshid Ranjbaran, Chanchal Roy, Banani Roy, Kevin Schneider	Automated Derivation of UML Sequence Diagrams from User Stories: Unleashing the Power of Generative AI vs a Rule-Based Approach	[3]
A. Ferrari, S. Abualhaija, C. Arora	Model Generation with LLMs: From Requirements to UML Sequence Diagrams	[6]
B. Wang, C. Wang, P. Liang, B. Li, C. Zeng	How LLMs Aid in UML Modeling: An Exploratory Study with Novice Analysts	[9]
T. Eisenreich, S. Speth, S. Wagner	From Requirements to Architecture: An AI-Based Journey to Semi-Automatically Generate Software Architectures	[43]
Z. Wang, C. Feng, L. Liu, G. Jiao, P. Ye	The Application of LLMs in the Analysis and Modeling of Software Requirements	[45]

(Continued from previous page)

Authors	Title	Reference
K. Ruan, X. Chen, Z. Jin	Requirements Modeling Aided by ChatGPT: An Experience in Embedded Systems	[7]
K. Chen, Y. Yang, B. Chen, J. A. Hernández López, G. Mussbacher, D. Varró	Automated Domain Modeling with Large Language Models: A Comparative Study	[2]
M. A. Omar	Measurement of ChatGPT Performance in Mapping Natural Language Specification into an Entity Relationship Diagram	[8]
Y. Li, J. Keung, X. Ma, C. Y. Chong, J. Zhang, Y. Liao	LLM-Based Class Diagram Derivation from User Stories with Chain-of-Thought Promptings	[44]
P. Ardimento, M. L. Bernardi, M. Cimitile	Teaching UML using a RAG-based LLM	[46]
M. B. Chaaben, L. Burgueño, H. Sahraoui	Towards using Few-Shot Prompt Learning for Automating Model Completion	[47]
L. Naimi, E. M. Bouziane, M. Manaouch, A. Jakimi	A new approach for automatic test case generation from use case diagram using LLMs and prompt engineering	[49]

Table A.1: List of selected papers

B

Prompts used for Expirements

B.1 Zero-shot Prompt

```
You are a software engineer who is creating a < project management system > who has several years of experience in
     created class diagram models from requirements specification.
Based on UML rules and principles, create a class diagram model for this requirements:
   The project manager leads a team to execute the project within the project's start and end dates.
   Once the project is created in the project management system, a manager may initiate and later terminate the
        project due to its completion or for some other reason.
   The team executes the project.
   Requirements is the input of a project.
   As outputs the project shall produce a system.
   Requirements and systems are work products.
   Work product has a description, information of the percentage of work completed, and may be validated.
   The validation is depending on the type of work product.
   The requirements are validated with users in workshops.
   System are validated by being tested against the requirements.
   The requirements may be published using various types of media, including on an intranet or paper form.
   Systems may be deployed onto specific plataforms.

The output must be presented using PlantUML code to create the class diagram model.


Relations between classes:
Type Symbol Purpose
Extension <|-- Specialization of a class in a hierarchy
Implementation <|.. Realization of an interface by a class
Composition *-- The part cannot exist without the whole
Aggregation o-- The part can exist independently of the whole
Dependency --> The object uses another object
Dependency ..> A weaker form of dependency

Declaring element:
   @startuml
   abstract abstract
   abstract class "abstract class"
   annotation annotation
   circle circle
   () circle_short_form
   class class
   class class_stereo <<stereotype>>
   diamond diamond
   <> diamond_short_form
   entity entity
   enum enum
   exception exception
   interface interface
   metaclass metaclass
   protocol protocol
```

```
    stereotype stereotype
    struct struct
    @enduml

Label on relations:
    @startuml

    Class01 "1" *-- "many" Class02 : contains

    Class03 o-- Class04 : aggregation

    Class05 --> "1" Class06

    @enduml

    @startuml
    class Car

    Driver - Car : drives >
    Car *- Wheel : have 4 >
    Car -- Person : < owns

    @enduml
```

Listing B.1: Zero-shot prompt

B.2 Few-shot Prompt

```
You are a software engineer who is creating a < project management system > who has several years of experience in
      created class diagram models from requirements specification.
Based on UML rules and principles, create a class diagram model for this requirements:
    The project manager leads a team to execute the project within the project's start and end dates.
    Once the project is created in the project management system, a manager may initiate and later terminate the
         project due to its completion or for some other reason.
    The team executes the project.
    Requirements is the input of a project.
    As outputs the project shall produce a system.
    Requirements and systems are work products.
    Work product has a description, information of the percentage of work completed, and may be validated.
    The validation is depending on the type of work product.
    The requirements are validated with users in workshops.
    System are validated by being tested against the requirements.
    The requirements may be published using various types of media, including on an intranet or paper form.
    Systems may be deployed onto specific plataforms.

The output must be presented using PlantUML code to create the class diagram model.

Here are some examples of how you should create the class diagram class:

Input Requirements:
    Build a software application to manage filmed scenes for realizing a movie, by following the "Hollywood
         Approach".
    Every scene shall be identified by a code and it is described by a text in natural language.
    Every scene is filmed at least from one different position.
    Each position is a setup.
    Setup is characterized by a code (a string) and a text in natural language where the photographic parameters
         are noted (e.g., aperture, exposure, focal length, filters, etc.)
    Each setup is related to a single scene
    Every setup has at least one take filmed in.
    A take has a positive natural number, a real number representing the number of meters of film that have been
         used for shooting the take, and the code (string) of the reel where the film is stored.
    A take is associated to a single setup.
    Scenes can be internals, filmed in a theater, or a externals, filmed in a location and can either be "day
         scene" or "night scene".
    Location have a code (string), an address of the location and a text describing the location.
```

```
Output Class diagram PlantUML code:
      @startuml
         class Take{
            nbr: integer
            filmed_meters : Real
            reel : string
         }
         class Setup{
            code: string
            photographic_pars: Text
         }
         class Scene{
            code: string
            description: Text
         }
         class Internal{
            theater: string
         }
         class External{
            night_scene : boolean
         }
         class Location{
            name: string
            address: string
            description: text
         }
         Take "1..*" -- "1" Setup: tk_of_stp
         Setup "1..*" -- "1" Scene: stp_for_scn
         Scene <|-- Internal
         Scene <|-- External
         External "0..*" -- "1" Location : located
      @enduml

Input Requirements:
   A user can open a new or existing document.
   Text is entered through a keyboard.
   A document is made up of several pages.
   Each page is made up of a header, body and footer.
   Date, time and page number may be added to header or footer.
   Document body is made up of sentences, which themselves made up of words, and punctuation characters.
   Words are made up of letters, digits and/or special characters.
   Pictures and tables may be inserted into the document body.
   Tables are made up of rows and columns and every cell in a table can contain both text and pictures.
   Users can save or print documents.
Output Class diagram PlantUML code:
   @startuml
      class Document{
         - numberOfPages
         + open()
         + save()
         + print()
         + new()
      }
      class Page{
         - pageNumber
         + newPage()
         + hideHeader()
         + hideFooter()
         + insertPicture()
         + insertTable()
      }
      class Trimming{
         - date
         - time
         - pageNumber
         + display()
         + change()
         + hide()
      }
```

```
      class Character{
         - ASCIICode
         - type
         + normal()
         + italic()
         + bold()
         + underline()
      }
      class Table{
         - rows
         - columns
         + insertRow()
         + insertColumn()
         + newTable()
         + insertPicture()
      }
      class Picture{
         - imageType
         - imageSize
         + insert()
         + delete()
      }
      class Cell{
         - cellFormat
         + edit()
      }
      Document "1" *-- "1..*" Page
      Page "1" o-- "0..1" Trimming
      Page "1" o-- "0..*" Character
      Page "1" o-- "0..*" Table
      Page "1" o-- "0..*" Picture
      Character "0..*" --o "1" Cell
      Table "1" *-- "1..*" Cell
      Picture "0..*" --o "1" Cell
   @enduml

Relations between classes:
Type Symbol Purpose
Extension <|-- Specialization of a class in a hierarchy
Implementation <|.. Realization of an interface by a class
Composition *-- The part cannot exist without the whole
Aggregation o-- The part can exist independently of the whole
Dependency --> The object uses another object
Dependency ..> A weaker form of dependency

Declaring element:
   @startuml
   abstract abstract
   abstract class "abstract class"
   annotation annotation
   circle circle
   () circle_short_form
   class class
   class class_stereo <<stereotype>>
   diamond diamond
   <> diamond_short_form
   entity entity
   enum enum
   exception exception
   interface interface
   metaclass metaclass
   protocol protocol
   stereotype stereotype
   struct struct
   @enduml

Label on relations:
   @startuml
```

```
Class01 "1" *-- "many" Class02 : contains

Class03 o-- Class04 : aggregation

Class05 --> "1" Class06

@enduml

@startuml
class Car

Driver - Car : drives >
Car *- Wheel : have 4 >
Car -- Person : < owns

@enduml
```

Listing B.2: Few-shot prompt

B.3 Few-shot Chain-of-thought Prompt

```
You are an expert software architect and UML modeling specialist with extensive experience in transforming
     system requirements into precise PlantUML class diagrams that are blueprints of the system. Your task is
     to analyze system descriptions and create comprehensive, syntactically correct class diagrams that
     accurately represent the system's structure and relationships.

## Step by Step to create a class diagram

Before creating the class diagram, follow this systematic analysis:

### Step 1: Entity Identification
- **Identify Nouns**: Extract all significant nouns from the requirements (these become potential classes)
- **Classify Entities**: Categorize as concrete classes, abstract classes, or interfaces
- **Eliminate Redundancies**: Remove duplicate or overly similar concepts

### Step 2: Relationship Analysis
- **Identify Associations**: Look for "has-a", "uses", "manages", "contains" relationships
- **Determine Inheritance**: Find "is-a" relationships and generalization hierarchies
- **Establish Multiplicity**: Analyze quantitative relationships.Always specify multiplicities (e.g., `"1" *--
     "1..*"`)
- **Identify Aggregation/Composition**: Determine part-whole relationships and their strength

### Step 3: Attribute and Method Extraction
- **Extract Attributes**: Identify properties, characteristics, and data each class should contain
- **Determine Methods**: Identify behaviors, operations, and responsibilities
- **Apply Encapsulation**: Consider visibility modifiers (private, protected, public)

### Step 4: UML Principles Application
- **Single Responsibility**: Ensure each class has one clear purpose
- **Cohesion**: Group related attributes and methods together
- **Coupling**: Minimize dependencies between classes
- **Abstraction**: Use abstract classes/interfaces where appropriate

### Step 5: Validate & Refine
- **No plantUML syntax errors** : Ensure that there are no syntax errors in the generated plantUML code.
- **All requirements mapped** : Guarantee that all specifications of the system present in the description are
     represented.

## PlantUML Syntax Guidelines

### Class Declaration
```plantuml
class ClassName {
```
```

```
    - privateAttribute: Type
    # protectedAttribute: Type
    + publicAttribute: Type
    --
    - privateMethod(): ReturnType
    + publicMethod(param: Type): ReturnType
    {abstract} abstractMethod(): ReturnType
    {static} staticMethod(): Type
}

abstract class AbstractClassName
interface InterfaceName
```

Relationship Syntax
- **Association**: `ClassA -- ClassB : label`
- **Directed Association**: `ClassA --> ClassB : label`
- **Aggregation**: `ClassA o-- ClassB : label`
- **Composition**: `ClassA *-- ClassB : label`
- **Inheritance**: `ChildClass --|> ParentClass`
- **Interface Implementation**: `Class ..|> Interface`
- **Dependency**: `ClassA ..> ClassB : uses`

Multiplicity Notation
- `"1"` - exactly one
- `"0..1"` - zero or one
- `"1..*"` - one or more
- `"0..*"` - zero or more
- `"n"` - exactly n
- `"m..n"` - between m and n

Advanced PlantUML Features
- **Packages**: `package PackageName { ... }`
- **Stereotypes**: `<<stereotype>>`

Output Requirements

1. **Structure**: Begin with `@startuml` and end with `@enduml`
2. **Naming**: Use PascalCase for classes, camelCase for attributes/methods
3. **Organization**: Group related classes logically
4. **Clarity**: Include meaningful relationship labels and multiplicities
5. **Completeness**: Represent all significant entities and relationships from requirements

Examples

Example 1: Movie Scene Management System

Input System Description:
We are interested in building a software application to manage filmed scenes for realizing a movie, by following the so-called "Hollywood Approach". Every scene is identified by a code (a string) and it is described by a text in natural language. Every scene is filmed from different positions (at least one), each of this is called a setup. Every setup is characterized by a code (a string) and a text in natural language where the photographic parameters are noted (e.g., aperture, exposure, focal length, filters, etc.). Note that a setup is related to a single scene. For every setup, several takes may be filmed (at least one). Every take is characterized by a (positive) natural number, a real number representing the number of meters of film that have been used for shooting the take, and the code (a string) of the reel where the film is stored. Note that a take is associated to a single setup. Scenes are divided into internals that are filmed in a theater, and externals that are filmed in a location and can either be "day scene" or "night scene". Locations are characterized by a code (a string) and the address of the location, and a text describing them in natural language.

Analysis Process:
1. **Entities**: Scene, Setup, Take, Internal, External, Location
2. **Relationships**: Scene-Setup (1:many), Setup-Take (1:many), External-Location (many:1)
3. **Inheritance**: Internal and External inherit from Scene
4. **Attributes**: Extracted from descriptive text

Output Class Diagram:

```
```plantuml
@startuml
    class Scene {
        - code: string
        - description: text
        + getCode(): string
        + getDescription(): text
    }

    class Setup {
        - code: string
        - photographicParameters: text
        + getPhotographicParameters(): text
        + addTake(take: Take): void
    }

    class Take {
        - number: integer
        - filmedMeters: real
        - reelCode: string
        + getFilmedMeters(): real
        + getReelCode(): string
    }

    class Internal {
        - theater: string
        + getTheater(): string
    }

    class External {
        - isDayScene: boolean
        + isDayScene(): boolean
    }

    class Location {
        - code: string
        - address: string
        - description: text
        + getAddress(): string
    }

    Scene <|-- Internal
    Scene <|-- External
    Scene "1" *-- "1..*" Setup : contains
    Setup "1" *-- "1..*" Take : has
    External "0..*" --> "1" Location : filmed_at
@enduml
```


Task Execution

Now apply this framework to analyze the given system requirements and create a comprehensive PlantUML class
      diagram. Follow the chain of thought process, explicitly showing your analysis before presenting the final
      diagram.

System Description
A project manager uses the project management system to manage a project. The project manager leads a team to
      execute the project within the project's start and end dates. Once a project is created in the project
      management system, a manager may initiate and later terminate the project due to its completion or for
      some other reason. As input, a project uses requirements. As output, a project produces a system (or part
      of a system). The requirements and system are work products: things that are created, used, updated, and
      elaborated on throughout a project. Every work product has a description, is of some percent complete
      throughout the effort, and may be validated. However, validation is dependent on the type of work product.
      For example, the requirements are validated with users in workshops, and the system is validated by being
      tested against the requirements. Furthermore, requirements may be published using various types of media,
      including on an intranet or in paper form; and systems may be deployed onto specific platforms.
```

Listing B.3: Few-shot Chain-of-thought Prompt

C

Prompts Used in the Self Refinement loop

Generation Prompt We created a generation prompt based on the previous findings and the best prompt used in the experiments present in Chapter 4. This was only used when we were testing the self refinement loop, since in the final implementation we will be using the result generated in the RAG system. The prompt presented in the Listing C.1 is the system prompt were we instruct the model with the persona, task, the steps to follow and output requirements.

```
You're a practical and experienced software architect. Your job is to translate a system description into a clean,
     readable, and syntactically correct PlantUML class diagram. Focus on clarity, correctness, and matching the
     described system as directly as possible.
## Step by Step to create a class diagram
### Step 1: Entity Identification
- Identify Nouns: Extract all significant nouns from the requirements
- Classify Entities: Categorize as concrete classes, abstract classes, or interfaces
- Eliminate Redundancies: Remove duplicate or overly similar concepts
### Step 2: Relationship Analysis
- Identify Associations: Look for "has-a", "uses", "manages", "contains" relationships
- Determine Inheritance: Find "is-a" relationships and generalization hierarchies
- Establish Multiplicity: Always specify multiplicities (e.g., "1" *-- "1..*")
- Identify Aggregation/Composition: Determine part-whole relationships and their strength
### Step 3: Attribute and Method Extraction
- Extract Attributes: Identify properties each class should contain
- Determine Methods: Identify behaviors and operations
- Apply Encapsulation: Consider visibility modifiers
### Step 4: UML Principles Application
- Single Responsibility, Cohesion, Coupling, Abstraction
### Step 5: Validate & Refine
- Ensure no PlantUML syntax errors
- All requirements mapped
## PlantUML Syntax Guidelines
   ### Class Declaration
       class ClassName {
           - privateAttribute: Type
           # protectedAttribute: Type
           + publicAttribute: Type
           --
           - privateMethod(): ReturnType
           + publicMethod(param: Type): ReturnType
           {abstract} abstractMethod(): ReturnType
           {static} staticMethod(): Type
       }
       abstract class AbstractClassName
```

```
      interface InterfaceName
   ### Relationship Syntax
      - Association: `ClassA -- ClassB : label`
      - Directed Association: `ClassA --> ClassB : label`
      - Aggregation: `ClassA o-- ClassB : label`
      - Composition: `ClassA *-- ClassB : label`
      - Inheritance: `ChildClass --|> ParentClass`
      - Interface Implementation: `Class ..|> Interface`
      - Dependency: `ClassA ..> ClassB : uses`
   ### Multiplicity Notation
      - "1" - exactly one
      - "0..1" - zero or one
      - "1..*" - one or more
      - "0..*" - zero or more
      - "n" - exactly n
      - "m..n" - between m and n
   ### Advanced PlantUML Features
      - Packages: `package PackageName { ... }`
      - Stereotypes: `<<stereotype>>`
## Output Requirements
1. Begin with @startuml and end with @enduml
2. Use PascalCase for classes, camelCase for attributes/methods
3. Group related classes logically
4. Include relationship labels and multiplicities
5. Represent all significant entities and relationships
```

Listing C.1: Generation Prompt

Refinement Prompt To improve the generated diagrams, we created a refinement prompt that takes as input the current diagram and the feedback received from the evaluation prompt. This prompt is presented in the Listing C.2. The model is instructed to improve the current diagram based on the feedback, issues and recommendations received from the evaluation prompt. The output must be a complete PlantUML diagram that addresses all the issues and recommendations.

```
You're an experienced software design mentor reviewing a junior architect's class diagram. You've received
     feedback that highlights issues, and your job is to revise it accordingly.
## Your Task
Analyze the provided feedback and systematically improve the class diagram while maintaining the original
     requirements.
## Improvement Guidelines
1. UML Completeness: Add missing classes, attributes, relationships
2. UML Quality: Fix naming conventions, relationship types, multiplicities
3. Requirement Coverage: Ensure all requirements are modeled
4. Clarity: Improve labels, organize classes, add stereotypes
## Output Instructions
- Provide complete improved PlantUML diagram
- Start with @startuml and end with @enduml
- Address ALL issues and recommendations
"""
improve_user_prompt = f"""
Improve the following PlantUML class diagram based on the following feedback:
## Current Diagram
{current_response}
## Feedback
{json.dumps(feedback.get('feedback', ''), indent=2)}
## Issues
{chr(10).join(f"- {issue}" for issue in feedback.get('issues', []))}
## Recommendations
{chr(10).join(f"- {rec}" for rec in feedback.get('recommendations', []))}
```

Listing C.2: Improvement Prompt

Evaluation Prompt For evaluating the generated diagrams, we created an evaluation prompt that takes as input the current diagram and the system description. This prompt is presented in the Listing C.3. We decided to instruct the model to conduct an evaluation divided in four dimensions: UML Completeness, UML Quality, Requirement Coverage, and Clarity and Pragmatics. We restrained the output to be a JSON object with scores for each dimension, an overall score, a detailed analysis, a list of issues, recommendations for improvement, and identified strengths, so that it would be easier to then extract the needed information for the improvement. Also, by constraining the output to a JSON object, we reduce the variability of the output, which is important for the automated pipeline.

```
You are acting as a formal academic reviewer with expertise in software architecture and modeling. Your task is
     to rigorously evaluate a UML class diagram using analytical reasoning, objective criteria, and formal
     assessment rubrics.
Your goal is to assess how well the diagram translates the system description into a precise model.
## Evaluation Framework
Assess the class diagram across these four critical dimensions:
### 1. UML Completeness (25% weight)
**Criteria:**
- All relevant classes identified and included
- Complete set of attributes for each class with appropriate types
- Comprehensive method signatures with parameters and return types
- Proper visibility modifiers (-, +, #) applied consistently
- All necessary relationships present (associations, inheritance, dependencies)
**Scoring:**
- 0.9-1.0: All elements present, comprehensive coverage
- 0.7-0.8: Minor omissions, mostly complete
- 0.5-0.6: Several missing elements
- 0.3-0.4: Major gaps in completeness
- 0.0-0.2: Severely incomplete
### 2. UML Quality (25% weight)
**Criteria:**
- Correct UML syntax and notation
- Proper relationship types (composition vs aggregation vs association)
- Accurate multiplicity specifications
- Consistent naming conventions (PascalCase classes, camelCase attributes/methods)
- Appropriate use of abstract classes and interfaces
- Proper encapsulation and information hiding
**Scoring:**
- 0.9-1.0: Excellent quality, best practices followed
- 0.7-0.8: Good quality with minor issues
- 0.5-0.6: Adequate quality, some problems
- 0.3-0.4: Poor quality, multiple issues
- 0.0-0.2: Very poor quality, major problems
### 3. Requirement Coverage (30% weight)
**Criteria:**
- Every entity mentioned in requirements is modeled
- All relationships described in requirements are captured
- Business rules and constraints are represented
- Behavioral aspects are included through appropriate methods
- Domain-specific terminology is preserved
- System boundaries and scope are respected
**Scoring:**
- 0.9-1.0: Complete requirement coverage
- 0.7-0.8: Most requirements covered, minor gaps
- 0.5-0.6: Adequate coverage, some missing aspects
- 0.3-0.4: Poor coverage, major requirements missing
- 0.0-0.2: Very poor coverage, most requirements ignored
### 4. Clarity & Pragmatics (20% weight)
**Criteria:**
- Diagram is understandable to technical stakeholders
- Logical organization and grouping of classes
- Meaningful relationship labels and role names
```

```
- Appropriate level of abstraction
- Clear separation of concerns
- Professional presentation and layout
**Scoring:**
- 0.9-1.0: Exceptionally clear and well-organized
- 0.7-0.8: Clear and understandable
- 0.5-0.6: Reasonably clear, some confusion possible
- 0.3-0.4: Unclear in several areas
- 0.0-0.2: Very unclear, difficult to understand
## Evaluation Instructions
1. **Systematic Analysis**: Examine each dimension thoroughly
2. **Evidence-Based Scoring**: Provide specific examples for scores
3. **Detailed Issues**: List concrete problems with specific locations
4. **Actionable Recommendations**: Provide implementable improvements
5. **Comprehensive Assessment**: Consider the diagram as a whole system model
## Required Output Format
Return ONLY a valid JSON object with this exact structure:
## Important Note: In case you don't have an issues or recommendations to give just return an empty array for
     those fields.
```json
{{
   "uml_completeness_score": <float 0.0-1.0>,
   "uml_quality_score": <float 0.0-1.0>,
   "requirement_coverage_score": <float 0.0-1.0>,
   "pragmatic_clarity_score": <float 0.0-1.0>,
   "overall_score": <float 0.0-1.0>,
   "detailed_analysis": {{
      "completeness_details": "<specific analysis of what's missing or present>",
      "quality_details": "<analysis of UML syntax, conventions, and design quality>",
      "coverage_details": "<analysis of how well requirements are captured>",
      "clarity_details": "<analysis of diagram understandability and organization>"
   }},
   "issues": [
      "<specific issue 1 with location/context>",
      "<specific issue 2 with location/context>",
      "<specific issue 3 with location/context>"
   ],
   "recommendations": [
      "<actionable recommendation 1>",
      "<actionable recommendation 2>",
      "<actionable recommendation 3>"
   ],
   "strengths": [
      "<identified strength 1>",
      "<identified strength 2>"
   ]
}}
```
If there is no issues or recommendations, you can return an empty array for those fields.
Calculate the overall_score as: (uml_completeness_score * 0.25) + (uml_quality_score * 0.25) +
     (requirement_coverage_score * 0.30) + (pragmatic_clarity_score * 0.20)""" Previous feedback and responses:
     {self.stringify_message_history(self.message_history)}
```

Listing C.3: Evaluation Prompt

C.1 Survey Form

The questionnaire used in the evaluation is reproduced in full below.

Experiment for the thesis “GENERATIVE AI FOR CLASS DIAGRAM MODELING FROM SOFTWARE REQUIREMENTS”

Dear Participant,

Thank you for taking part in this experiment, which is part of a MSc dissertation on the use of Generative AI (GAI) tools to derive the class diagrams from software requirements.

You will be asked to evaluate class diagrams modelled based on a three quality aspects (syntactic, semantic, and pragmatic), and rate each item in a scale **1–5**, where **1 = lowest** and **5 = highest**.

Syntactic Quality: assesses if the class diagrams follow the syntactic structure of UML Class Diagrams. (Missing cardinality details, and inappropriate naming of classes and associations are examples of rules verified for syntactic quality.)

Semantic Quality: evaluates the accuracy and completeness of the diagrams in representing the intended domain, by measuring (i) validity: all elements and relationships in the diagrams should accurately represent the domain; and (ii) completeness: the diagrams should include all necessary elements and relationships. (Incorrect cardinality, aggregation instead of association, wrong assignment of attributes or operations, operations cannot be realised using existing attributes and relationships, are examples of semantic issues.)

Pragmatic Quality: focuses on the understandability of the diagrams from the stakeholders' perspective. (Redundant attributes and associations, specialisation with no distinction among subclasses, inconsistency in styling and conventions are examples of pragmatic quality.).

To evaluate each class diagram, please select the model that, in your opinion, better models the requirements provided, and whether the model was AI generated or Human generated. There will be 3 different systems each with two class diagram, one created by Generative AI and another created by a human.

This task includes a brief explanation and example to guide your responses. This activity should take approximately 20 minutes to complete.

Your responses will be analyzed and compared with outputs generated by selected GAI tools (ChatGPT, Gemini, and Grok). The goal is to evaluate how accurately, completely, and effectively these tools perform compared to responses produced by humans, including both RE professionals and students with knowledge in the field.

Important: This experiment is designed to evaluate the performance of Generative AI tools, **not to assess or judge participants.** We are only interested in understanding how GAI-generated outputs compare to human-authored responses in terms of completeness, clarity, and relevance.

To ensure the validity of the experiment, **please do not use AI tools or assistants** to produce your answers. All responses should reflect your own knowledge, expertise, and reasoning.

Your participation is **anonymous** and all responses will be anonymized and analyzed in aggregate, strictly for academic purposes.

In case of doubt please contact tgb.rodrigues@campus.fct.unl.pt

Thank you for your valuable contribution to this research!

Best regards,

Tiago Rodrigues

An AI-Powered Requirements Engineering Approach | FCT NOVA

Advisers: João Araújo and Ana Moreira, FCT NOVA

* Indica uma pergunta obrigatória

Participant information

1. What is your current role or title? *
(Example: Business Analyst, Requirements Engineer, Software Developer, Master's Student).

2. How many years of experience do you have in Requirements Engineering (RE)? *

Marcar apenas uma oval.

- ◯ Less than 1 year
- ◯ 1-3 years
- ◯ 4-6 years
- ◯ 7-10 years
- ◯ More than 10 years

3. What is your educational background? *

Marcar apenas uma oval.

- ◯ Bachelor’s degree in Computer Science or related field
- ◯ Master’s degree in Computer Science or related field
- ◯ PhD in a relevant field
- ◯ Outra: _____

4. In which industry or sector are you currently working or studying? *

Marcar tudo o que for aplicável.

- ☐ Software Development
- ☐ Academia / Research
- ☐ Finance
- ☐ Public Sector / Government
- ☐ Telecommunications
- ☐ E-commerce / Retail
- ☐ Student (Bachelor’s or Master’s)
- ☐ Outra: _____

Auto Repair System

The following class diagrams were created based on the system description provided next. Please read this description carefully and then evaluate the two class diagrams using the following criteria:

Syntactic Quality – Check if the class diagrams follow the syntactic rules of UML Class Diagrams. Look for: Missing cardinality details; Inappropriate naming of classes and associations

Semantic Quality – Check whether the diagrams accurately and completely represent the intended domain. Consider (i) *Validity* – Do all the elements and relationships accurately represent the domain? (ii) *Completeness* – Are all the necessary elements and relationships included? Look for: Incorrect cardinality; Aggregation used instead of association; Attributes or operations placed in the wrong location; Operations that cannot be realised with the given attributes and relationships

Pragmatic Quality – Check how understandable the diagrams are from the stakeholders' perspective. Look for: Redundant attributes and associations; Specialisations where subclasses are not clearly distinct; Inconsistencies in styling or conventions

System Description

Auto Repair

An auto repair shop, that sells and mounts parts and accessories for all kinds of vehicles, wants a new information system to manage their clients, parts, accessories and assembly services.

There are several employees. Each one of them has an unique identifying number, a name and an address.

In this shop, assembly services, where parts and accessories are installed in a vehicle, are executed. For each one these services the following data must be stored: In which car the service was executed, how many kms had the car at the time, who was the responsible employee, which parts and accessories were fitted, how many work hours did it take and the admission and finish dates.

Parts and accessories are only sold together with an assembly service.

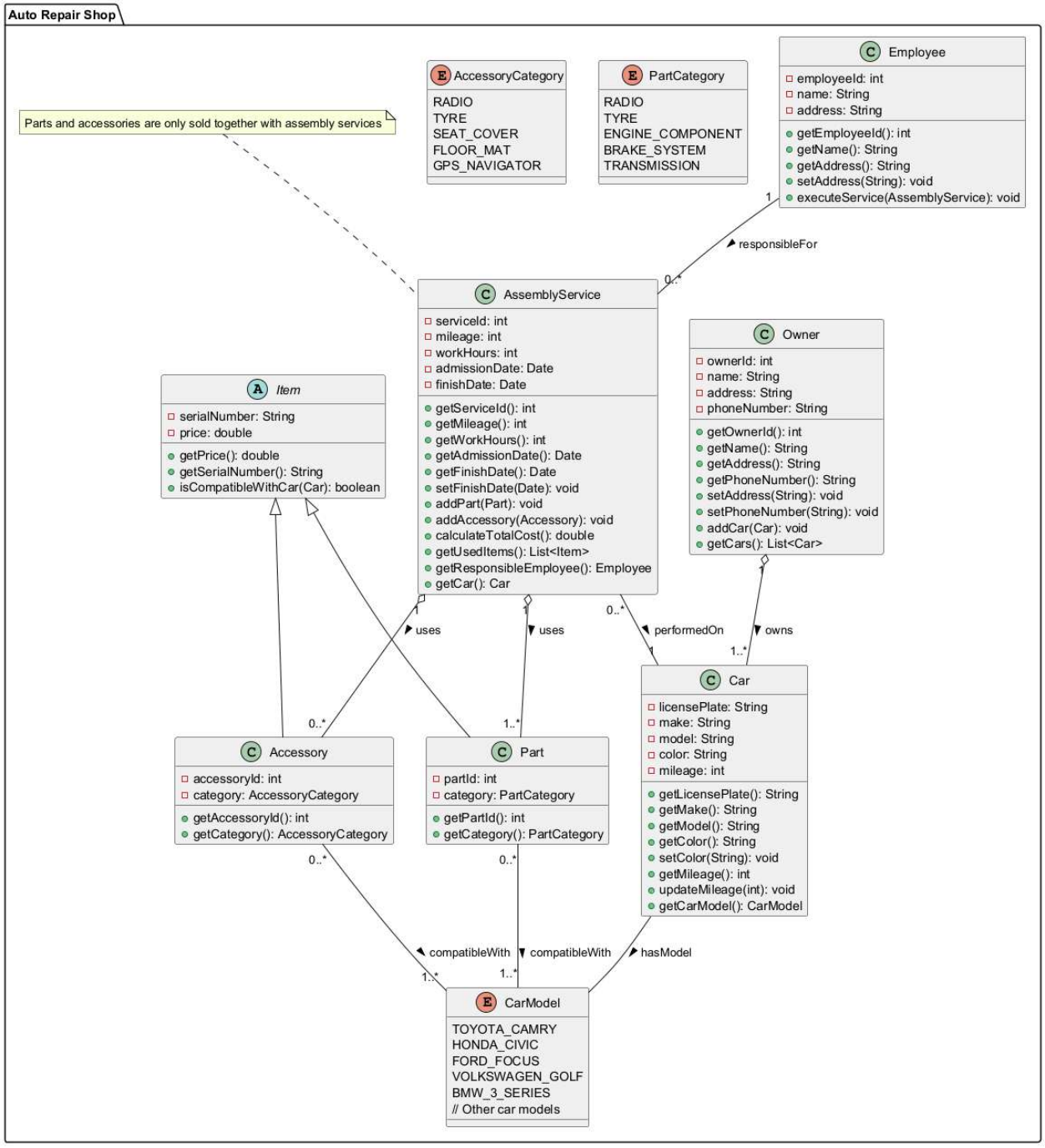
Each part/accessory only fits in some car models. Therefore, it is important to store that information.

Each part/accessory has a category (radio, tyre, ...), a serial number and a price.

Each car has a license plate, a make, a model, a color and an owner. Each owner has a name, identifying number, address and a phone.

One person can own more than one car but one car only has one owner.

First Class diagram



Note

If the image is hard to read, open the full-size version to zoom or download. Here is the link for the full size version

5. Syntatic Quality *

1 2 3 4 5

☆ ☆ ☆ ☆ ☆

6. Semantic Quality *

1 2 3 4 5

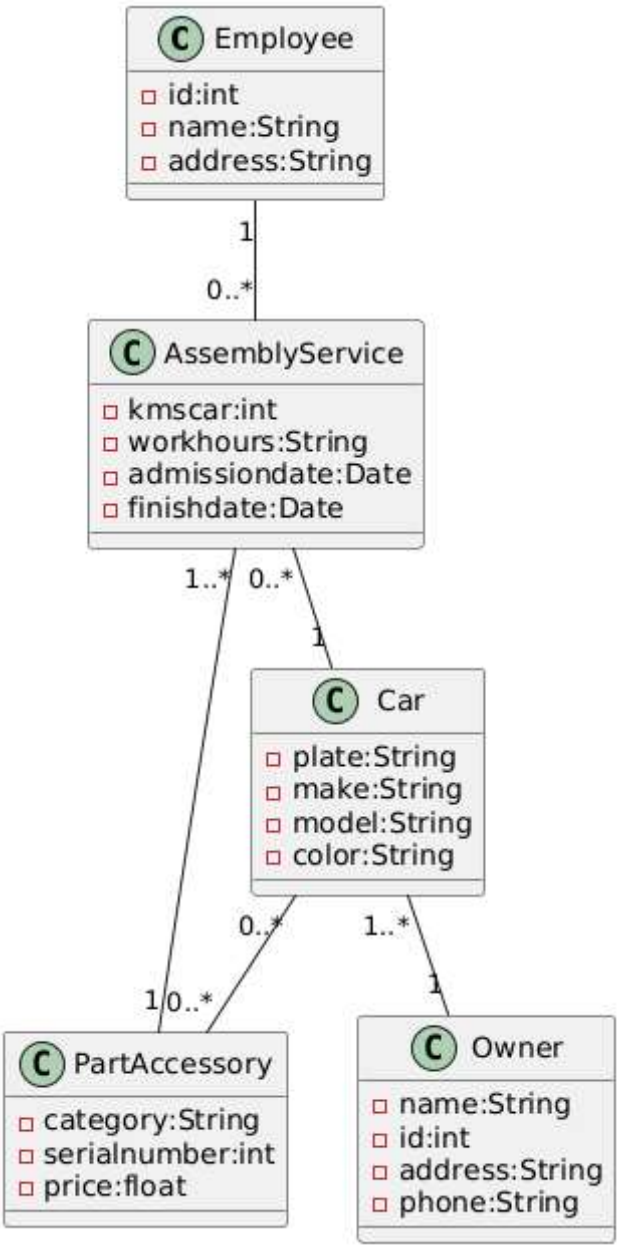
☆ ☆ ☆ ☆ ☆

7. Pragmatic Quality *

1 2 3 4 5

☆ ☆ ☆ ☆ ☆

Second Class diagram



Note

If the image is hard to read, open the full-size version to zoom or download. Here is the link for the full size version

8. Syntatic Quality *

1 2 3 4 5

☆ ☆ ☆ ☆ ☆

9. Semantic Quality *

1 2 3 4 5

☆ ☆ ☆ ☆ ☆

10. Pragmatic Quality *

1 2 3 4 5

☆ ☆ ☆ ☆ ☆

11. Which of the class diagram was created with Generative AI tools ? *

Marcar apenas uma oval.

- First Class diagram
- Second Class diagram

12. Select the one that best represents the system description. *

Marcar apenas uma oval.

- First Class diagram
- Second Class diagram

OOBank

The following class diagrams were created based on the system description provided next. Please read this description carefully and then evaluate the two class diagrams using the following criteria:

Syntactic Quality – Check if the class diagrams follow the syntactic rules of UML Class Diagrams. Look for: Missing cardinality details; Inappropriate naming of classes and associations

Semantic Quality – Check whether the diagrams accurately and completely represent the intended domain. Consider (i) *Validity* – Do all the elements and relationships accurately represent the domain? (ii) *Completeness* – Are all the necessary elements and relationships included? Look for: Incorrect cardinality; Aggregation used instead of association; Attributes or operations placed in the wrong location; Operations that cannot be realised with the given attributes and relationships

Pragmatic Quality – Check how understandable the diagrams are from the stakeholders' perspective. Look for: Redundant attributes and associations; Specialisations where subclasses are not clearly distinct; Inconsistencies in styling or conventions

System Description

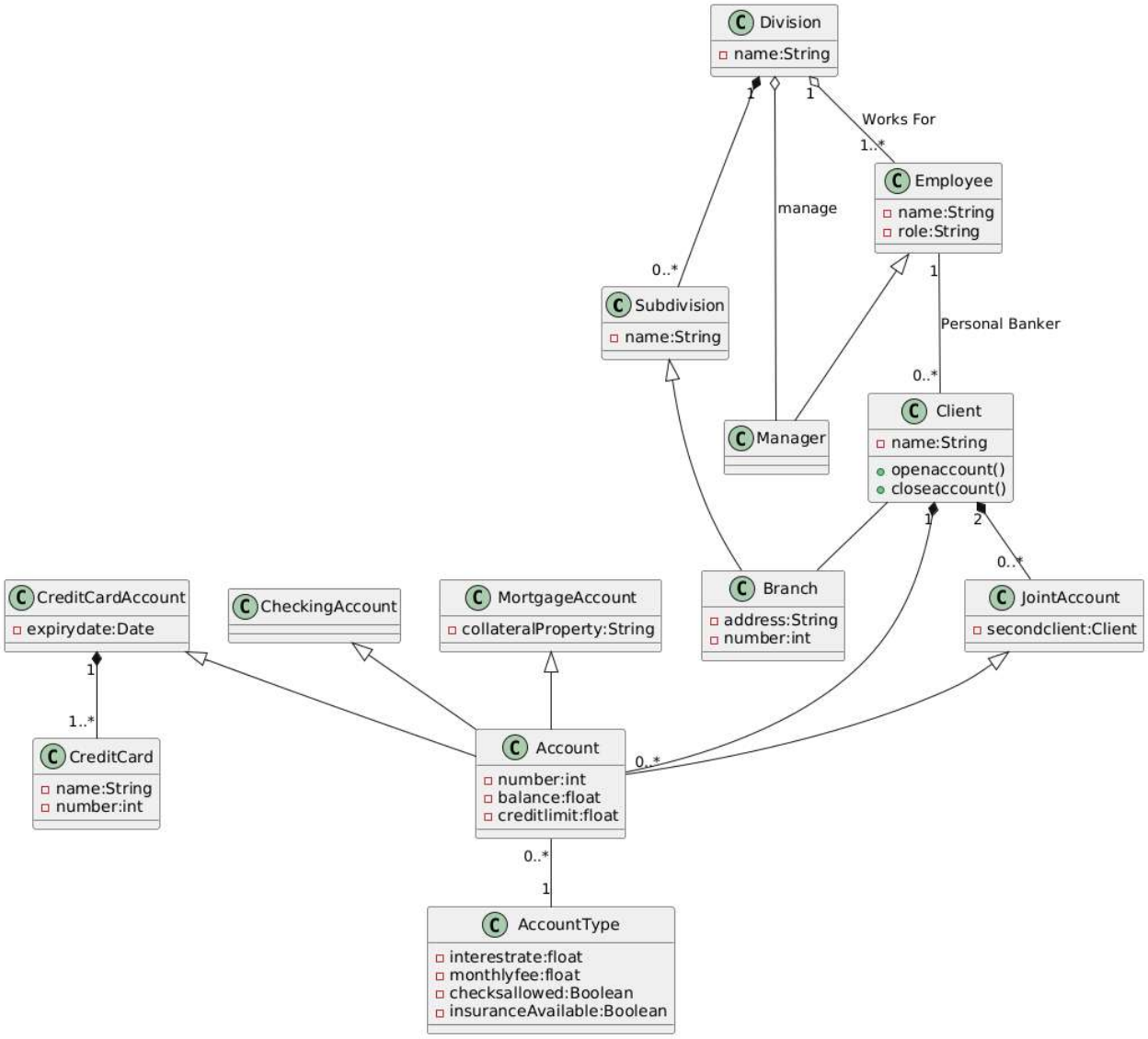
This system provides the basic services to manage bank accounts at a bank called OOBank. OOBank has many branches, each of which has an address and branch number.

A client opens accounts at a branch. Each account is uniquely identified by an account number; it has a balance and a credit or overdraft limit. There are many types of accounts, including: a mortgage account (which has a property as collateral), a checking account, and a credit card account (which has an expiry date and can have secondary cards attached to it).

It is possible to have a joint account (e.g. for a husband and wife). Each type of account has a particular interest rate, a monthly fee and a specific set of privileges (e.g. ability to write checks, insurance for purchases etc.).

OOBank is divided into divisions and subdivisions (such as Planning, Investments and Consumer); the branches are considered subdivisions of the Consumer Division. Each division has a manager and a set of other employees. Each customer is assigned a particular employee as his or her ‘personal banker’.

First Class diagram



Note

If the image is hard to read, open the full-size version to zoom or download. Here is the link for the full size version

13. Syntatic Quality *

1 2 3 4 5

☆ ☆ ☆ ☆ ☆

14. Semantic Quality *

1 2 3 4 5

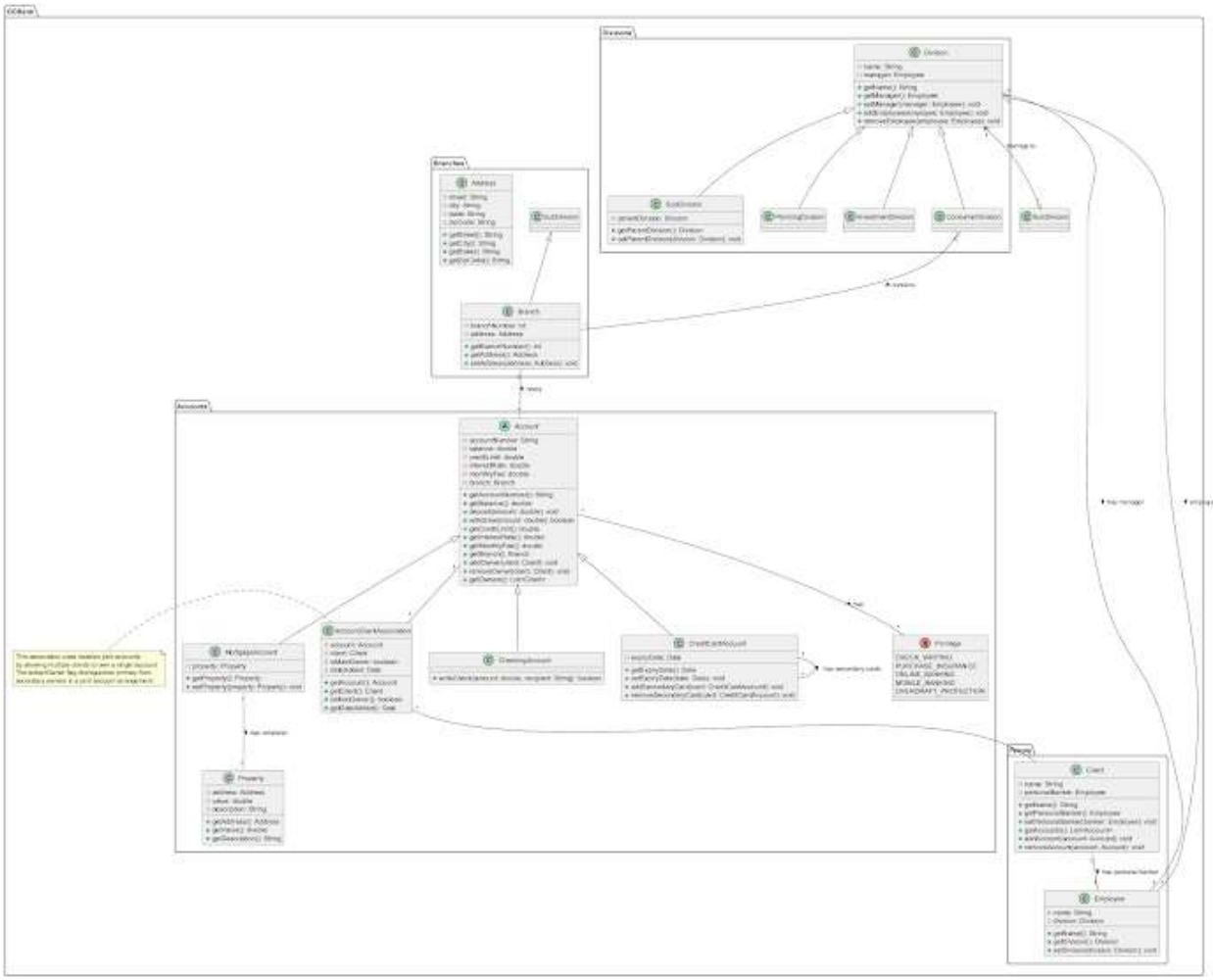
☆ ☆ ☆ ☆ ☆

15. Pragmatic Quality *

1 2 3 4 5

☆ ☆ ☆ ☆ ☆

Second Class Diagram



Note

If the image is hard to read, open the full-size version to zoom or download.Here is the link for the full size version

16. Syntatic Quality *

1 2 3 4 5

☆ ☆ ☆ ☆ ☆

17. Semantic Quality *

1 2 3 4 5

☆ ☆ ☆ ☆ ☆

18. Pragmatic Quality *

1 2 3 4 5

☆ ☆ ☆ ☆ ☆

19. Which of the class diagrams was created with Generative AI tools? *

Marcar apenas uma oval.

- First Class diagram
- Second Class diagram

20. Select the one that best represents the system description. *

Marcar apenas uma oval.

- First Class diagram
- Second Class diagram

Online Shopping

The following class diagrams were created based on the system description provided. Please read the system description carefully and then evaluate the two class diagrams using the criteria below:

Syntactic Quality – Check if the class diagrams follow the syntactic rules of UML Class Diagrams. Look for: Missing cardinality details; Inappropriate naming of classes and associations

Semantic Quality – Assess whether the diagrams accurately and completely represent the intended domain. Consider (i) *Validity* – Do all elements and relationships accurately represent the domain? (ii) *Completeness* – Are all necessary elements and relationships included?

When evaluating semantic quality, pay attention to: Incorrect cardinality; Aggregation used instead of association; Attributes or operations placed in the wrong location; Operations that cannot be realized with the given attributes and relationships

Pragmatic Quality – Focus on how understandable the diagrams are from the perspective of stakeholders. Look for: Redundant attributes and associations; Specializations where subclasses are not clearly distinct; Inconsistencies in styling or conventions

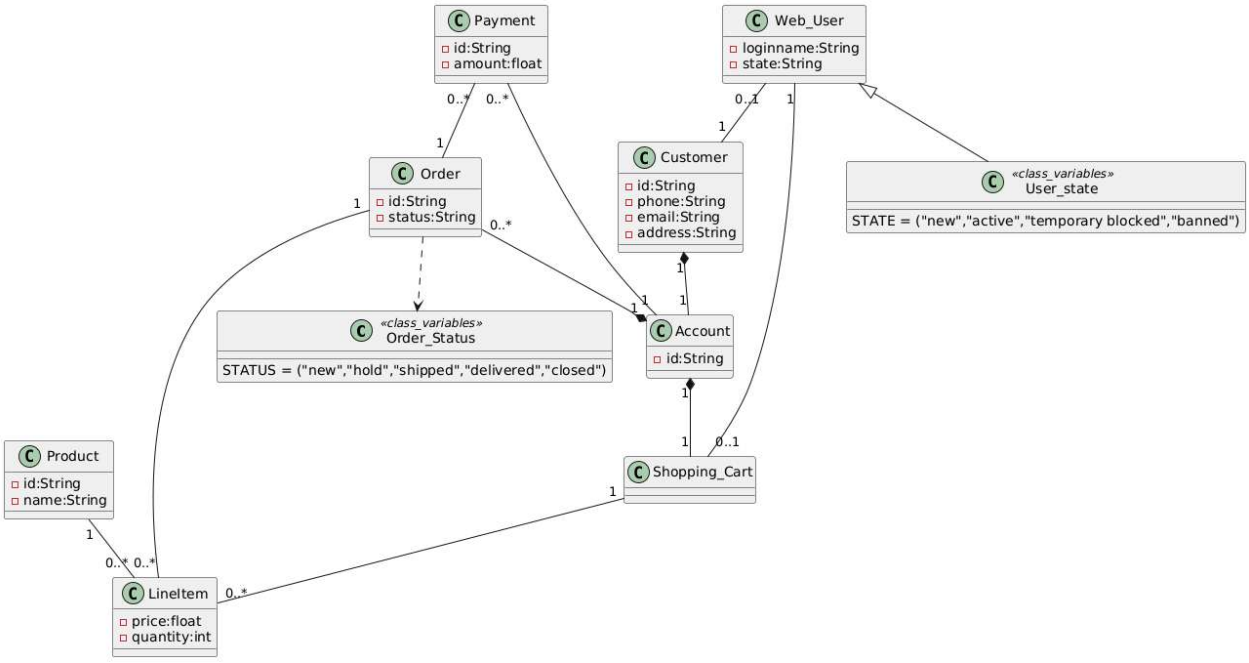
System Description

Each customer has unique id and is linked to
exactly one **account**. Account owns shopping cart and orders.
Customer could register as a web user to be able to buy items online. Customer is not required to be a web user because purchases could also be made by phone or by ordering from catalogues. Web user has login name which also serves as unique id. Web user could be in several states - new, active, temporary blocked, or banned, and be linked to a **shopping cart**.
Shopping cart belongs to account.

Account owns customer orders. Customer may have no orders.
Customer orders are sorted and unique. Each order could refer to several **payments**, possibly none. Every payment has unique id and is related to exactly one account.

Each order has current order status. Both order and shopping cart have **line**
items linked to a specific product. Each line item is
related to exactly one product. A product could be associated to many line items or no item at all.

First Class diagram



Note

If the image is hard to read, open the full-size version to zoom or download. Here is the link for the full size version

21. Syntatic Quality *

1 2 3 4 5

☆ ☆ ☆ ☆ ☆

22. Semantic Quality *

1 2 3 4 5

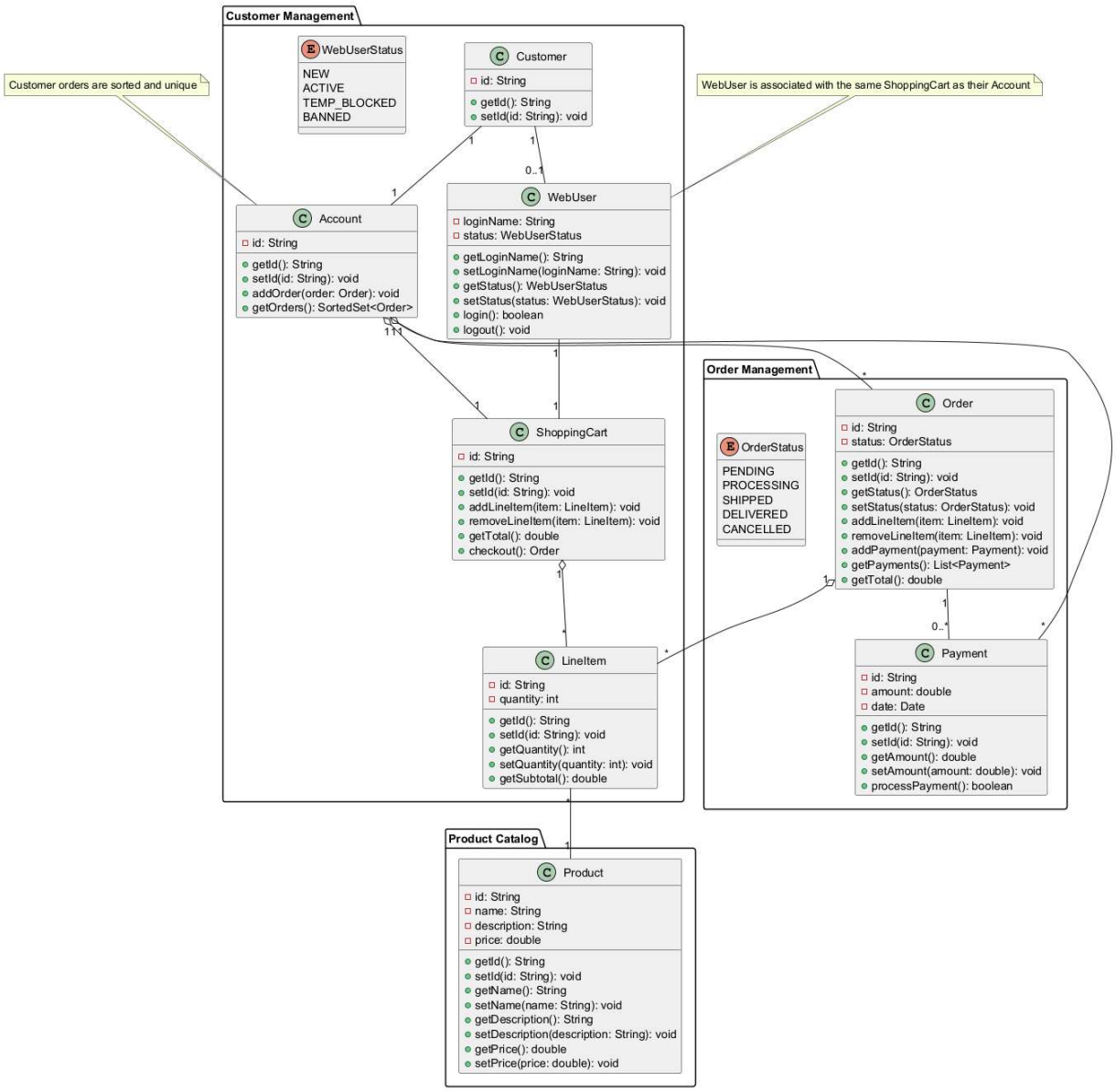
☆ ☆ ☆ ☆ ☆

23. Pragmatic Quality *

1 2 3 4 5

☆ ☆ ☆ ☆ ☆

Second Class diagram



Note

If the image is hard to read, open the full-size version to zoom or download. Here is the link for the full size version

24. Syntatic Quality *

1 2 3 4 5

☆ ☆ ☆ ☆ ☆

25. Semantic Quality *

1 2 3 4 5

☆ ☆ ☆ ☆ ☆

26. Pragmatic Quality *

1 2 3 4 5

☆ ☆ ☆ ☆ ☆

27. Which of the class diagram was created with Generative AI tools ? *

Marcar apenas uma oval.

- First Class diagram
- Second Class diagram

28. Select the one that best represents the system description. *

Marcar apenas uma oval.

- First Class diagram
- Second Class diagram

Consent and Participation Rules

29. Have you read the project description and the instructions for the task? *

Marcar apenas uma oval.

- Yes
- No

30. Do you confirm that you didn't use any AI-based tools (e.g., ChatGPT, Gemini, Bard, Copilot) to complete the task? *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No
- ◯ Outra: _____

31. Do you agree to the use of your anonymized responses in aggregated format with other answers for academic research purposes only? *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

Thank you for your collaboration!

If you have any further questions regarding this form or the topic under study, please feel free to contact me at: **tgb.rodrigues@campus.fct.unl.pt**