POLITECNICO DI TORINO

Master degree course in Engineering and Management

Master Degree Thesis

Large Language Models for Requirements Engineering

Supervisor
prof. Riccardo Coppola

Candidate
Stefano Rosa

Academic Year 2024-2025

Summary

The following research explores the integration of Large Language Models (LLMs) into Goal-Oriented Requirements Engineering (GORE) within software development, with a particular emphasis on Requirements Engineering (RE). By leveraging transformer-based architectures such as Generative Pre-trained Transformers (GPT), in particular the one developed by Open AI, Chat GPT, the research investigates their potential to address key challenges in requirements elicitation, analysis, and specification. Specifically, it examines how LLMs could be fundamental in facilitating the automated extraction of stakeholder goals from textual requirements documentation, enhancing the systematic derivation of system objectives.

Furthermore, the research evaluates the role of LLMs in aligning high-level system goals with the specifications of Representional State Transfer (REST) Application Programming Interfaces (APIs). By employing LLMs for sub-goal analysis and API mapping, it becomes possible to develop novel methodologies for bridging the gap between user-centric requirements and technical implementation, thereby promoting consistency and coherence throughout the software development lifecycle.

Contents

List of Tables

List of Figures

Part I

Opening section

Chapter 1

Introduction

1.1 Overview of the problem

Software development involves translating high-level goals and requirements into concrete implementation details, often realized through the use of Application Programming Interfaces (APIs).
In this scenario, bridging the gap between abstract requirements and specific API functions poses significant challenges to developers. A potential solution could be found in the realm of **Large Language Models** (LLMs).
One of the main goals of this research is to explore the multifaceted nature of this problem, by identifying key factors that contribute to the gap and examining the LLM strategy to mitigate its impact.

As a starting point, our objective is to investigate the genesis and fundamental factors that contribute to this divergence. The gap mentioned above arises from a multitude of interconnected variables that complicate the translation of abstract requirements into concrete implementations.

Firstly, **ambiguity** in requirements specifications. The high-level goals provided by the client often lack the precision and granularity required for direct translation into specific API functions. [1]
Ambiguous requirements can result in misinterpretations and divergent implementations, leading to inefficiencies and inconsistencies in the software development process. In a scientific paper conducted by Nuseibeh and Easterbrook (2000), it is highlighted that requirements are often written in natural language, which is inherently ambiguous and prone to multiple interpretations. In this scenario, stakeholders may express requirements in broad terms, leaving significant room for developers' interpretations, which can lead to different implementations of the same requirement. [2]
This ambiguity is further exacerbated by the fact that stakeholders have different perspectives and priorities, resulting in discrepancies in both their understanding and articulation of requirements.
A closer examination of the research conducted by Nuseibeh and Easterbrook, combined with insights from additional sources, reveals that ambiguity is a significant challenge across various project domains. The following bar chart illustrates that government and public sector projects experience the highest levels of ambiguity (70%), while embedded systems, although still concerning, exhibit relatively lower levels of ambiguity of around

50%. [11]
It is crucial to note that high rates of ambiguity often lead to project delays and cost overruns, underscoring the need for more effective requirements engineering practices.

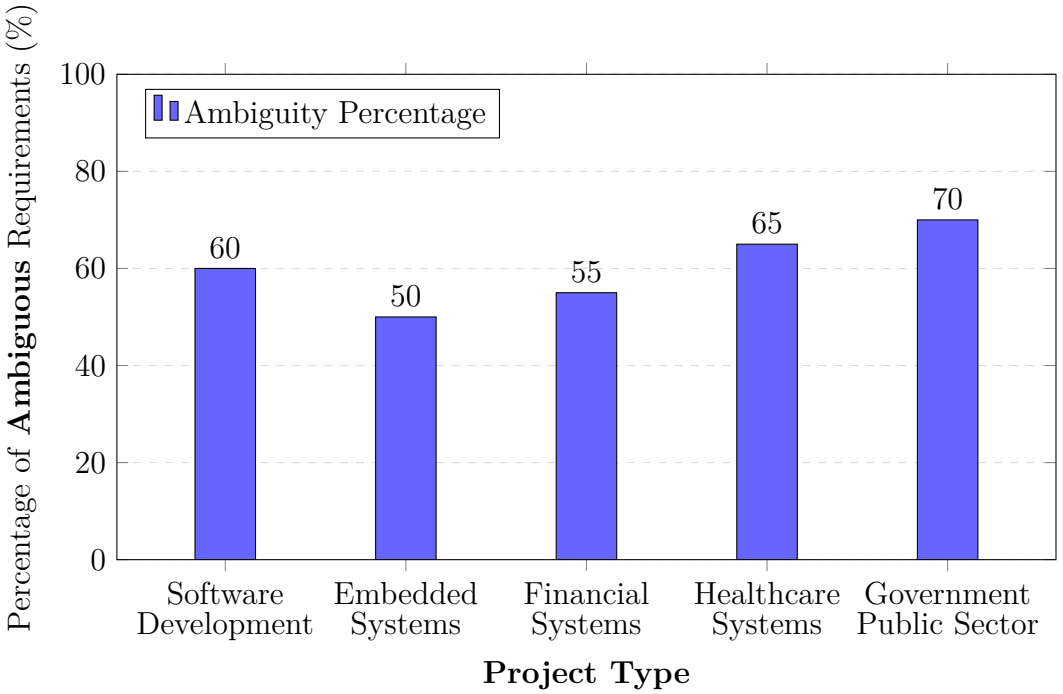

Figure 1.1: Percentage of ambiguous requirements across different project domains.

Another threat can be depicted is the **variability** that characterizes the API functionality. APIs typically offer a wide range of functions to accomplish several tasks, and selecting the most appropriate for a given requirement can not be so trivial.
In this context, developers must navigate through a plethora of options, taking into account factors such as performance, compatibility, and maintainability, in order to make informed decisions. This decision-making process can be particularly daunting when dealing with APIs that offer overlapping functionalities or when documentation is not comprehensive or up-to-date.
Moreover, the wide range of functionalities provided by modern APIs can overwhelm developers, making it difficult to choose the best function for a specific requirement. [3]

In addition to this, **the dynamic nature** of APIs also represents a challenge. APIs evolve over time, introducing more complexity as new functions are added, deprecated, or simply modified. This continuous evolution requires developers to constantly update their knowledge and adapt their previous implementations, which can be resource-intensive and error-prone. [4]

Drawing from the study *"How do developers react to API evolution? A large-scale empirical study,"* published in the Software Quality Journal [5], the following line chart

illustrates the increasing percentage of deprecated API methods over a five-year period. This trend underscores the dynamic nature of APIs and the challenges developers face in maintaining up-to-date applications, showing in percentage a linear increase in API deprecation. The study analyzed the *Pharo* ecosystem, which includes approximately 3.600 distinct systems over six years, to observe the evolution of API and its impact. The findings revealed that as APIs evolve, client developers often need to adapt to new functionalities, and client systems may require modifications to align with updated APIs. This continuous evolution requires that developers remain vigilant and proactive in updating their applications to ensure compatibility and functionality.

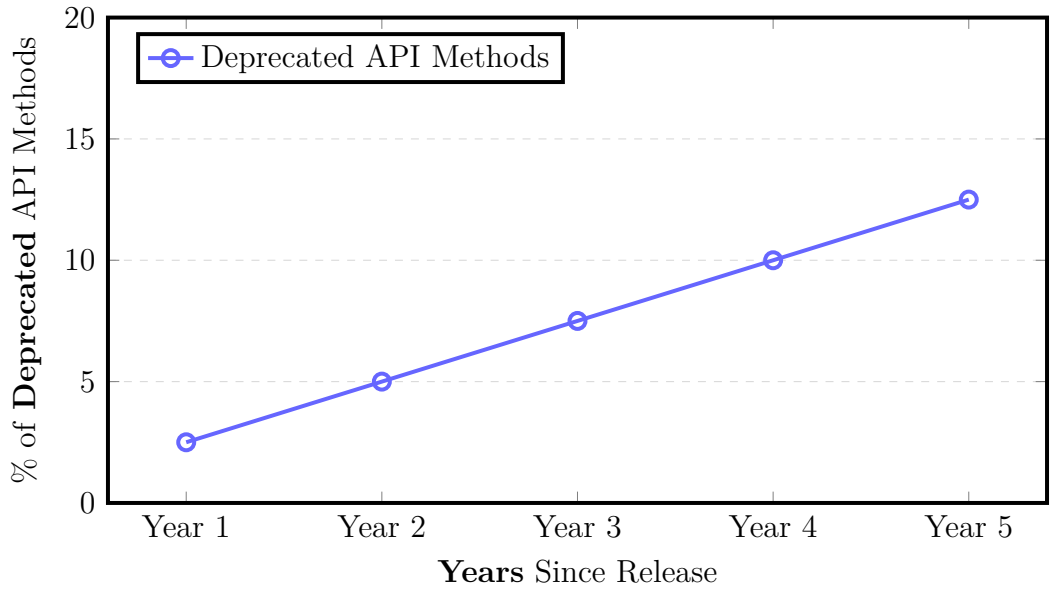

Figure 1.2: Trend of API methods deprecation over five years.

Finally, the **mismatch** in domain technology compounds the challenge of bridging the gap: the domain-specific jargon used by stakeholders to describe high-level goals and requirements can be vastly different from the terminology used in API documentation and programming paradigms. This discrepancy, as for the ambiguity previously described, can result in miscommunications between stakeholders, developers, and API users, which hinders effective collaboration and problem solving, leading to suboptimal API implementations. [6]

In this scenario, a fundamental question spontaneously arises: How is it possible to effectively **bridge** this gap?
This question is the crucial one that drives our research.

1.2 Bridging the gap

Software engineering practitioners employ a variety of strategies aimed at enhancing **requirements** elicitation, analysis, and specification processes.

The first and most widely used strategy can be seen in the adoption of **goal-oriented requirement engineering** (GORE) methodologies.
In general, goal-oriented approaches facilitate a holistic understanding of the needs and preferences of stakeholders, allowing a more structured and coherent translation of high-level goals into specific API calls. By focusing on the underlying goals and objectives driving the software development effort, practitioners find a way to prioritize requirements and identify the most relevant API functions to achieve them, breaking down broad goals into more manageable subgoals and tasks, allowing easier mapping to API calls. [3]
This strategy could be considered as the fundamental one used in our research.

A complementary approach to GORE methodologies is the use of **model-driven** development techniques, which provide a structured and systematic methodology. This strategy involves creating formal models that encapsulate the fundamental aspects of the problem domain, enabling developers to clearly define and analyze the relationships between high-level goals, system requirements, and API functions. By offering a visual and structured representation of these elements, model-driven development helps bridge the gap between abstract specifications and concrete implementations.
Moreover, the most significant advantage of this approach lies in its capacity to support iterative refinement and continuous validation of requirements. Through this iterative process, stakeholders can actively participate in reviewing the evolving system model, providing feedback at each stage, which in turn improves their understanding of the expected behavior of the system and ensures that requirements remain aligned with business and user needs. [7]

In addition, another contribution to our research environment is given by **domain-specific languages** (DSLs). These languages provide significant advantages in software development by offering customized solutions for specific problem domains.
Unlike general-purpose programming languages such as Java or Python, which are designed for broad applicability, domain-specific languages are crafted to address the unique requirements, constraints, and semantics of a particular application domain. This specialization allows DSLs to serve as powerful tools for translating abstract requirements into concrete API functions with greater precision. In this scenario, one of their most valuable contributions lies in their ability to minimize ambiguity in requirement specifications and facilitate seamless alignment between domain knowledge and technological implementation. By reducing the cognitive gap between stakeholders and developers, DSLs help ensure that software solutions remain closely aligned with domain-specific needs, ultimately improving efficiency and maintainability.

Ultimately, the integration of **natural language processing** (NLP) techniques and LLMs offers a promising avenue to automate the mapping of requirements to API. By training LLMs on a large corpus of software documentation and API usage examples, developers can leverage these models to automate the process of this mapping.

In particular, NLP is a field of artificial intelligence and computational linguistics concerned with enabling computers to understand, interpret, and generate human language

in a way that is both meaningful and useful. In the context of this research, LLMs can be used to parse high-level requirement descriptions and suggest relevant API functions, thus streamlining the development process and reducing the room for human error.

An important example is that LLMs can analyze textual requirements and automatically generate code snippets or API calls that match the described functionality, significantly speeding up the development process while improving accuracy. [8]

A comprehensive study titled "*An Analysis of Ambiguity Detection Techniques for Software Requirements Specification (SRS)*" by Khin Hayman Oo et al. [19] explores the prevalence and challenges associated with ambiguity in Software Requirements Specifications (SRS) and examines several techniques for its detection, showing the potential of NLP techniques within this context. Ambiguity is important to acknowledge in the scenario of SRS since it can lead to misinterpretations, inconsistencies, and costly errors in software development. The study categorized the ambiguity detection methods into three primary approaches.

1. **Manual Approach**
   This traditional technique relies on human reviewers who carefully examine SRS documents to identify ambiguous terms, inconsistent language, and contradictions. Reviewers manually inspect the text, applying their expertise to detect vague or unclear statements that could lead to misunderstandings during implementation. Although this method allows for nuanced interpretation, it is time-consuming and prone to human error.

2. **Semi-Automatic Approach Using Natural Language Processing (NLP)**
   In this method, NLP tools help human reviewers detect ambiguous terms and structures. These tools are designed to process natural language within the Software Requirements Specification, highlighting potential areas of ambiguity by analyzing syntax, semantics, and contextual meaning.
   Although not fully automated, this approach takes advantage of the capabilities of NLP algorithms to enhance the accuracy and speed of ambiguity detection.

3. **Semi-Automatic Approach Using Machine Learning**
   This technique incorporates machine learning algorithms, such as Naïve Bayes text classification, to autonomously identify ambiguous statements within the SRS. By training the machine learning model on a dataset of labeled ambiguous and non-ambiguous phrases, the algorithm learns to predict ambiguity in new, unseen SRS documents. This approach significantly reduces human intervention and can scale effectively with larger datasets.

The study provides a quantitative evaluation of these approaches, highlighting their respective strengths and drawbacks.
The Naïve Bayes classifier demonstrated strong performance, achieving a **precision** of 85% and a **recall** of 78 %, where:

- **Precision** refers to the proportion of correctly identified ambiguous statements out of all statements flagged as ambiguous, which means that 85% of the detected ambiguities were actual ambiguities.
- **Recall** measures the proportion of actual ambiguous statements correctly identified by the classifier, indicating that it successfully detected 78% of all ambiguities present in the dataset.

On the other hand, the manual review process, although still effective, achieved a lower precision of 70% and required significantly more time and effort.
This underscores the trade-off between accuracy and efficiency: manual reviews allow for thorough examination, while machine learning-based and NLP-assisted approaches offer faster, more scalable, and more automated solutions for detecting ambiguity in SRS. [19]

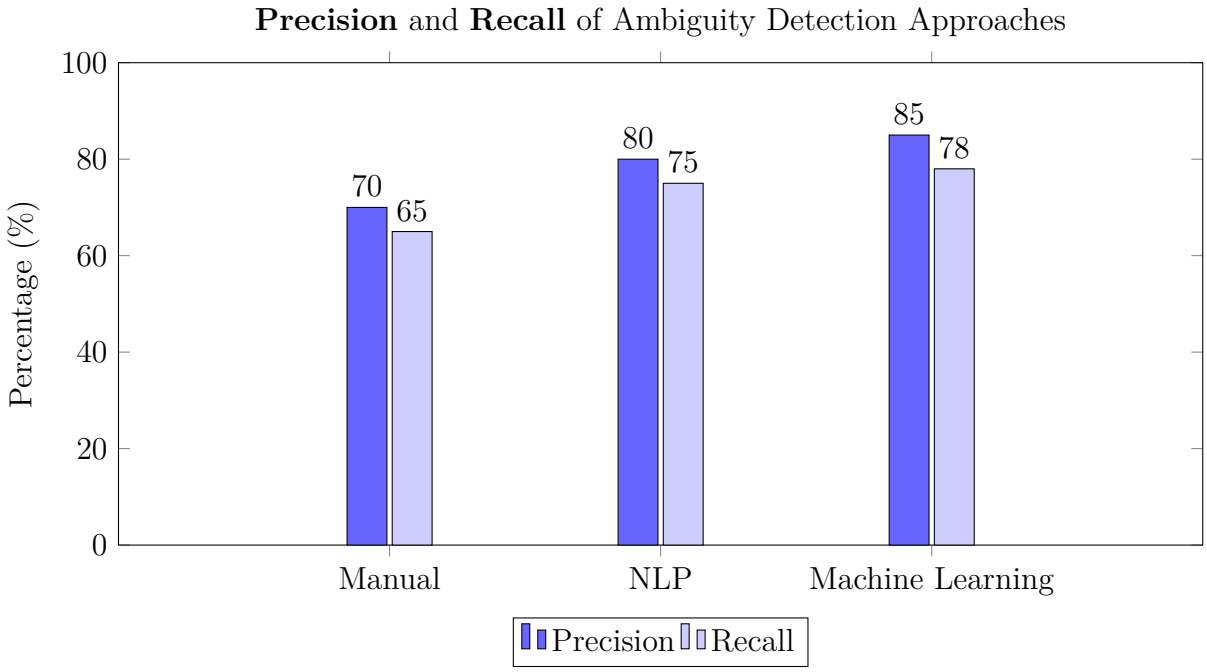

Figure 1.3: Comparison of ambiguity detection approaches in SRS.

Chapter 2

Background and related work

2.1 Literature review on LLMs in software development

The progress of **Generative Artificial Intelligence** (GAI) in recent years has catalyzed a paradigm shift in the field of software engineering, particularly in the domain of Requirements Engineering (RE).
Generative Artificial Intelligence, encompassing a spectrum of Artificial Intelligence techniques focused on autonomously generating new content, has emerged as a transformative force that paves the way for reshaping traditional software development practices.

At the forefront of this technological revolution are the already-depicted **Large Language Models** (LLMs). As already stated, Large Language Models are sophisticated AI systems endowed with extraordinary linguistic capabilities, which enable them to comprehend, generate, and manipulate human-like text with unprecedented fluency and coherence. [7]

LLMs, exemplified by landmark technologies such as the **Generative Pre-trained Transformer** (GPT) series, have become compulsory tools in automating various software development tasks, including natural language understanding, code generation, documentation synthesis, and software testing. [9]
The GPT series, developed by OpenAI, has set a new benchmark in the field of NLP by demonstrating how pre-trained transformer architectures can perform a wide array of language-related tasks with minimal fine-tuning. [10]

The remarkable capabilities of these models are underpinned by the attention mechanism, which allows them to focus on different parts of the input text dynamically, thus capturing complex dependencies and context.

These advancements have enabled LLMs to tackle sophisticated language tasks, such as **few-shot learning**, where the model can adapt to new tasks with only a few examples, showcasing their versatility and robustness in handling diverse linguistic challenges. The implications of these capabilities are extensive, as they offer new methodologies for addressing long-standing issues in software engineering, such as the ambiguity and inconsistency of software requirements. [7]

On the other hand, the field of **Requirements Engineering** (RE) is crucial to the successful delivery of software systems, which includes activities such as generating, analyzing, specifying, and validating requirements.
Here, the gap becomes evident: Traditional RE processes often suffer from inefficiencies, ambiguities, and inconsistencies in requirement specifications. These challenges can lead to project delays, increased costs, and compromised software quality.

Moreover, the inherent complexity of modern software systems, coupled with the dynamic nature of stakeholder requirements, exacerbates these challenges, and requires more robust and adaptive approaches to RE. [2]

In this scenario, Large Language Models present a promising solution, since they offer a diverse array of practical applications in RE, addressing challenges across the entire RE lifecycle.

During **requirement elicitation**, LLMs can analyze stakeholder communications, user feedback, and domain-specific documents to extract implicit requirements and identify potential conflicts or inconsistencies. This capability is particularly valuable in environments where requirements are continuously evolving, as it ensures that all relevant information is captured and appropriately addressed. Furthermore, in this phase, LLMs can help synthesize various inputs into cohesive requirement statements, thereby improving the clarity and precision of the elicitation process. [12]

In **requirement analysis**, LLMs can categorize requirements, extract domain concepts, and identify relationships between requirements and system functionalities. Using natural language processing capabilities, LLMs can perform automated analysis of requirements to uncover hidden dependencies and ensure comprehensive coverage of all necessary aspects. This automated analysis is instrumental in maintaining the integrity of requirements throughout the project lifecycle, reducing the risk of overlooked or misunderstood requirements. [13]

In addition, LLMs can significantly improve the **specification** of requirements by generating clear, precise, and unambiguous statements from high-level descriptions. They can also help validate requirements by comparing them with predefined criteria or existing documentation to ensure consistency and completeness. [14]
This validation process is crucial to mitigate the risk that errors propagate through subsequent stages of the software development lifecycle.

Going deeper in the analysis, LLMs can also support the **maintenance** and **evolution** of requirements by automatically updating documentation in response to changes in stakeholder needs or system functionalities. This capability ensures that the requirements specification remains relevant and aligned with the evolving project context.

Another important aspect to be taken into consideration is the fact that the integration of LLMs into Requirement Engineering processes can facilitate enhanced **traceability** and **manageability** of requirements. That means that LLMs can map requirements to design documents, code modules, and test cases, providing a comprehensive traceability matrix that supports impact analysis and change management. This traceability is of great importance, since it is vital to ensure that all changes in requirements are systematically reflected across all related artifacts, thereby maintaining coherence and consistency

throughout the software development process. [15]

By leveraging all the capabilities described above, the ability to improve the efficiency and accuracy of Requirements Engineering processes arises, enhancing the overall quality and success of software development projects.

This technological advancement not only mitigates traditional challenges in RE but also opens new avenues for innovative software engineering practices. As LLMs continue to evolve, their integration into RE processes is expected to become more sophisticated, leading to further improvements in software development outcomes. The ongoing research and development in this field highlights the potential of LLMs to drive the significant advancements in the way requirements are managed and implemented in software projects, paving the way for more agile and responsive development methodologies.

2.2 The integration of LLMs in RE: a case study

A research review titled "*Generative AI for Requirements Engineering: A Systematic Literature Review*" analyzed 105 articles about requirements engineering published between 2019 and 2024. [20]
This review explores the methods and techniques used in **Generative Artificial Intelligence (GAI)**, specifically Large Language Models, within the field of RE, which corresponds to our field of scope. The review highlights the practical impact of LLMs in automating and improving several phases of the RE lifecycle.

One of the critical findings of the review involves **BERT**. BERT (Bidirectional Encoder Representations from Transformers) is a type of pre-trained deep learning model used primarily for Natural Language Processing (NLP) tasks. Developed by Google, it represents a significant advancement in the field of NLP due to its ability to understand the context of words in a sentence in both directions: left-to-right and right-to-left, hence the term "bidirectional."
This study focused particularly on the **requirement specification phase**, showing how BERT can automatically identify missing terminologies in software specifications that could otherwise be overlooked. The results revealed a significant increase in the completeness of requirement documents due to the model's ability to identify missing terms and concepts crucial for comprehensive requirements.

In particular, the research presented quantitative data that compare the percentage of missing terminology identified in the requirements documents before and after the application of BERT.
Before using BERT, a significant portion (30%) of terminology was left unaddressed, which could potentially lead to ambiguities or incomplete specifications.

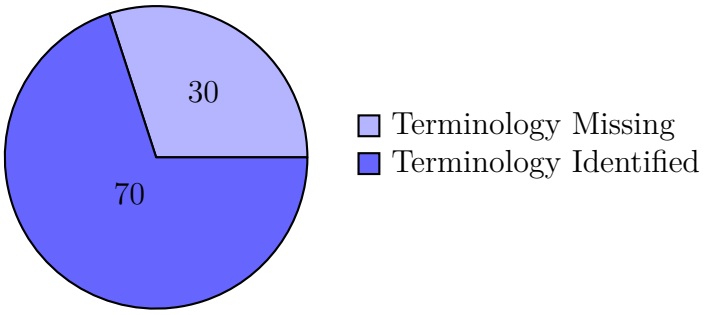

Figure 2.1: Identification of missing terminology before LLMs implementation.

After BERT was integrated into the process, this number decreased significantly. Only 9% of the terminology remained unaddressed, with 21% of the previously missing terminology identified, demonstrating the potential power of LLMs in improving the completeness of requirements.

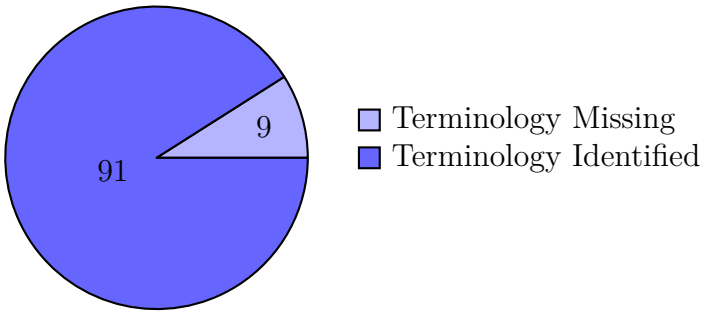

Figure 2.2: Identification of missing terminology after LLMs implementation.

These findings reinforce the potential of LLMs to support RE professionals by automating tasks that traditionally require profound manual effort.
Although human expertise remains essential for validation and refinement, AI-driven models offer a scalable, efficient, and highly accurate approach to refining requirement documents.

2.3 The Role of LLMs in Software Engineering

The field of **language processing** has undergone significant transformations over the past decades, driven by advancements in computations power, machine learning techniques, and access to large-scale datasets. Traditional **Language Models** (LMs) have served as the foundation for text generation and comprehension, allowing several Natural Language Processing (NLP) applications. However, the gradual appearance of **Large Language Models** has redefined the landscape, introducing models with billions of parameters that can understand, generate, and manipulate human language with unprecedented accuracy. Unlike earlier models, LLMs are trained on expansive and diverse corpora, allowing them to capture intricate patterns in linguistic structures and simulate human-like text generation. These models, including transformer-based architectures such as BERT, GPT-3, and GPT-4, have demonstrated exceptional proficiency in several NLP tasks, including text summarization, question answering, sentiment analysis, and contextual understanding. By learning from vast textual datasets, LLMs are progressively bridging the gap between human language and machine-generated language, making them invaluable tools for researchers exploring the nuances of human communication.

2.3.1 The impact of LLMs on software engineering

Alongside advancements in language processing, Software Engineering is increasingly leveraging innovations driven by large language models (LLMs). Traditionally, SE tasks have relied on structured programming paradigms, manual code reviews, and extensive documentation to ensure software quality. However, LLMs have introduced a paradigm shift by demonstrating their capability to automate and enhance critical aspects of software development. One of the key reasons for this transformation is that many SE challenges can be framed as text-based problems. Since programming languages share syntactic similarities with natural language, LLMs can be leveraged for a wide range of SE tasks, including:

- **Code summarization** - Automatically generate high-level descriptions of code functionality.
- **Code completion** - Predict and suggest the next lines of code based on context.
- **Bug detection and debugging** - Identify potential errors, while simultaneously proposing fixes.
- **Code translation** - Convert code between different programming languages.
- **Natural-language-to-code (NL2Code) tasks** - Generate executable code from human-readable descriptions.

The effectiveness of LLMs in these areas has been demonstrated through several empirical studies. In a study conducted in 2021, named "*Evaluating Large Language Models Trained on Code*", the power of LLMs within the coding environment had been investigated. In particular, **Codex**, a LLM with 12 billion parameters, has exhibited remarkable proficiency in handling complex coding tasks, solving 72.31% of Python programming challenges formulated by human evaluators. This highlights the practical utility of LLMs in real-world software engineering scenarios, where automation and efficiency are critical. [21]

2.3.2 The choice of GPT-4 interface

Among the various LLMs available, GPT-4 from OpenAI has been selected as the primary model for this research due to its state-of-the-art performance, versatility, and robustness in handling SE tasks. Several factors contributed to this decision:

1. **Enhanced language understanding and reasoning**
   GPT-4 exhibits improved contextual understanding, reasoning capabilities, and coherence in text generation compared to its predecessors. This is particularly valuable in SE tasks that require a deep understanding of technical documentation, code semantics, and software requirements.

2. **Superior performance in code-related tasks**
   OpenAI has demonstrated that GPT-4 significantly outperforms previous models in code generation, debugging, and optimization. Its ability to process complex programming logic makes it an ideal tool for automating SE workflows.

3. **Scalability and adaptability**
   Unlike domain-specific models, the GPT interface is a general-purpose LLM that can be fine-tuned or adapted to specific SE tasks. This flexibility allows for a broader range of applications, from requirements analysis to automated documentation generation.

4. **Robustness in handling natural language and code**
   Many SE tasks require bridging the gap between natural language and programming languages (e.g., converting natural language specifications into executable code). GPT-4 excels in NL2Code tasks, making it a powerful asset for requirement engineering and automated software development.

5. **Empirical validation**
   Prior research and industry applications have showcased GPT's effectiveness in SE, with companies integrating it into development pipelines for tasks like automated code review, documentation generation, and AI-assisted coding. Its real-world applicability further validates its relevance to this study.

Given these advantages, GPT-4 served as the foundational model for this research, providing a comprehensive framework to explore the integration of LLMs in Requirements Engineering.

2.4 Semantic API alignment as a solution

One innovative approach leveraging Large Language Models within the context of requirements engineering is the **Semantic API Alignment** (SEAL) framework.
SEAL aims to bridge the gap between high-level user goals and specific Application Programming Interface (API) functions by automatically aligning them with appropriate API calls through semantic similarity and content-aware analysis. This methodology enhances the efficiency and accuracy of software development processes, particularly within Goal-Oriented Requirements Engineering, where aligning goals with technical specifications is crucial for successful project outcomes. [16]
A distinctive feature of the SEAL framework is its ability to analyze not only the literal terms but also the contextual meaning and intent behind the requirements under scope. This capability allows the framework to identify more relevant and contextually appropriate API functions, thereby reducing the risk of mismatches and ensuring that the underlying needs of stakeholders are met effectively.

2.4.1 Core mechanisms of SEAL

The SEAL framework is built upon several key mechanisms that facilitate the alignment process:

1. **Goal analysis and decomposition**
   LLMs are employed to analyze high-level user goals and decompose them into subgoals or tasks. This decomposition involves breaking down complex requirements into manageable components that can be directly linked to specific API functions. For instance, a high-level goal such as "enhance user authentication" can be decomposed into two subgoals like "implement two-factor authentication" and "integrate biometric verification," each of which can be mapped to specific API calls. This step provides a detailed roadmap for implementation that aligns technical solutions with business objectives.

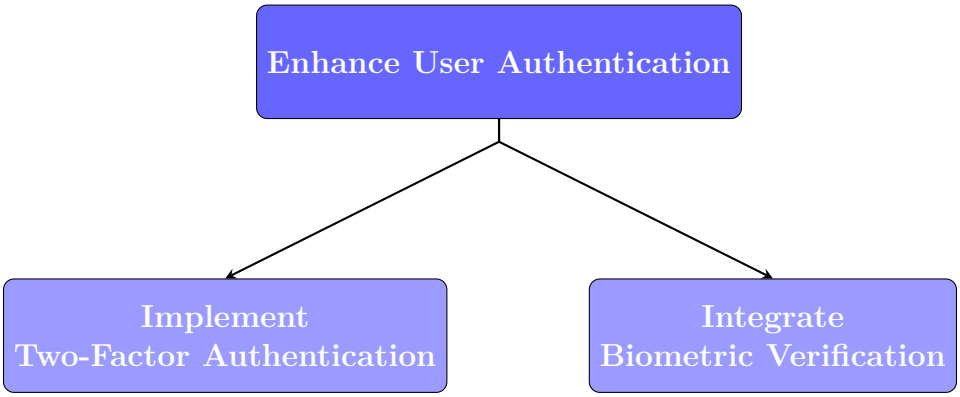

Figure 2.3: Decomposition of high-Level Goal into subgoals.

2. **Semantic Linking and Mapping**
   Once the subgoals are identified, LLMs map these tasks to appropriate API functions by understanding the semantics of both the requirements and the APIs. This semantic mapping involves deep learning models trained on an extensive corpus of software documentation and API usage examples, enabling precise and contextually relevant matches between requirements and API functions.
   Advanced techniques such as embedding representations and transformer architectures are employed to capture the nuances of both natural language requirements and technical API descriptions. Embedding representations allow LLMs to encode semantic information in a high-dimensional space, facilitating the detection of similarities and relationships between requirements and API functions. This approach leverages the power of transformer-based models, which have revolutionized natural language processing by enabling deep contextual understanding and capturing complex dependencies in text.

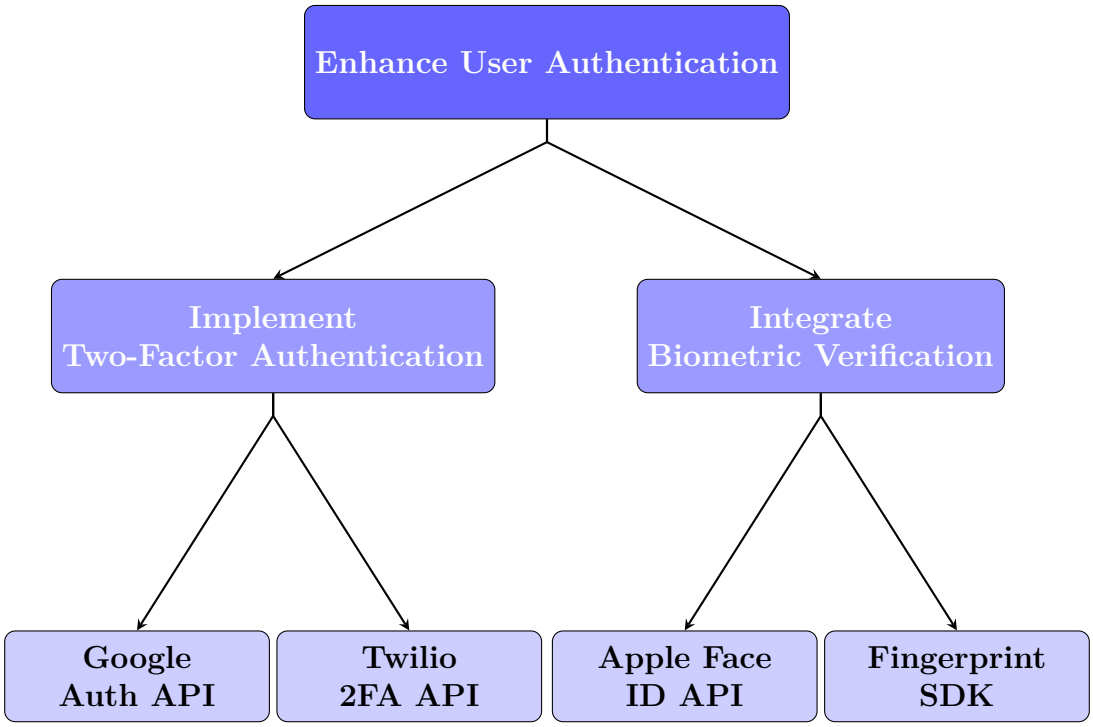

Figure 2.4: Mapping of low-level goals to API functions.

3. **Iterative refinement and validation**
   SEAL employs an iterative approach according to which initial mappings are validated and refined through feedback loops involving stakeholders, developers, and the model GPT itself.
   This iterative process is fundamental in fine-tuning the alignment and addressing any discrepancies or ambiguities. Continuous feedback ensures that the mappings remain both accurate and aligned with evolving user expectations and technical constraints. On the other hand, the input coming from the developers assists the validation of the technical feasibility of the suggested API functions.
   This iterative refinement is supported by advanced techniques such as:

- **Active learning**
  The model incrementally improves its performance by incorporating new data and feedback over time.
- **Self-critique mechanism**
  This is a prompt engineering technique, which is going to be used later in the research and involves prompting the model to evaluate its own output, identify errors or weaknesses, and refine its response accordingly. Instead of just generating a response, the model is explicitly instructed to critically evaluate the output for coherence, bias, or other issues and then to revise and improve the response based on its own critique.

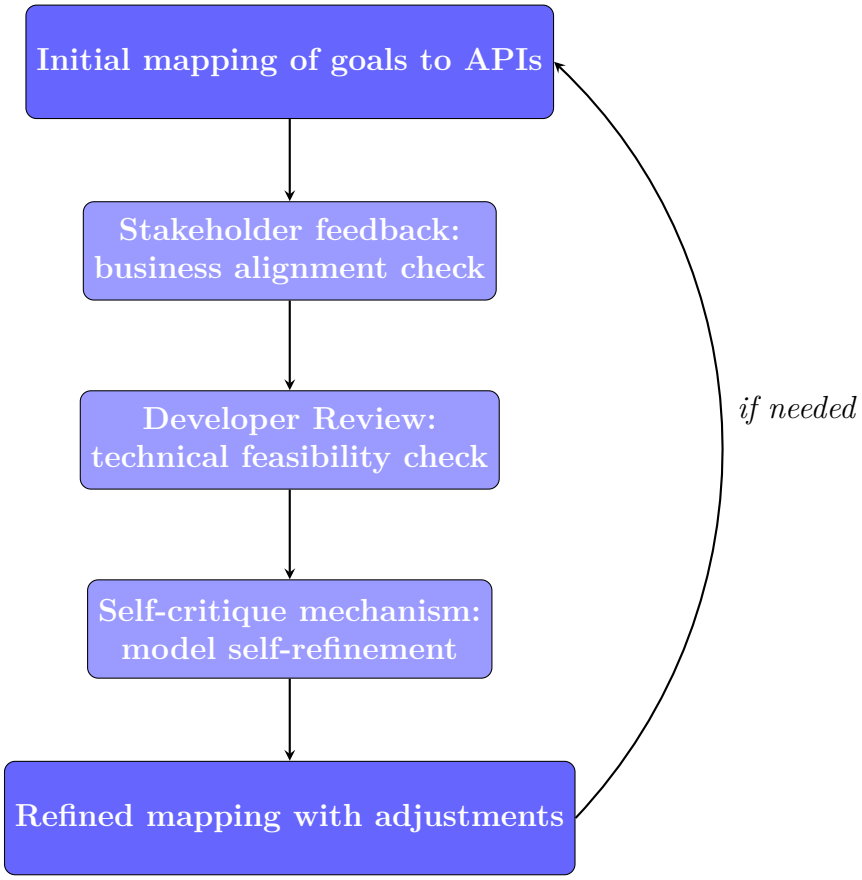

Figure 2.5: Iterative refinement process in the SEAL framework.

4. **Dynamic adaptation to API evolution**
   APIs are dynamic and often evolve over time. SEAL includes mechanisms to adapt to these changes by updating the mappings as new API functions are added or existing ones are modified. This dynamic adaptation is facilitated by periodic retraining of LLMs on updated datasets, ensuring that the mappings remain relevant and accurate despite changes in the API landscape. For example, if a new API function is introduced that provides a more efficient way to achieve a certain sub-goal, SEAL can quickly adapt and update its mapping to utilize this new function, as described in the following diagram.

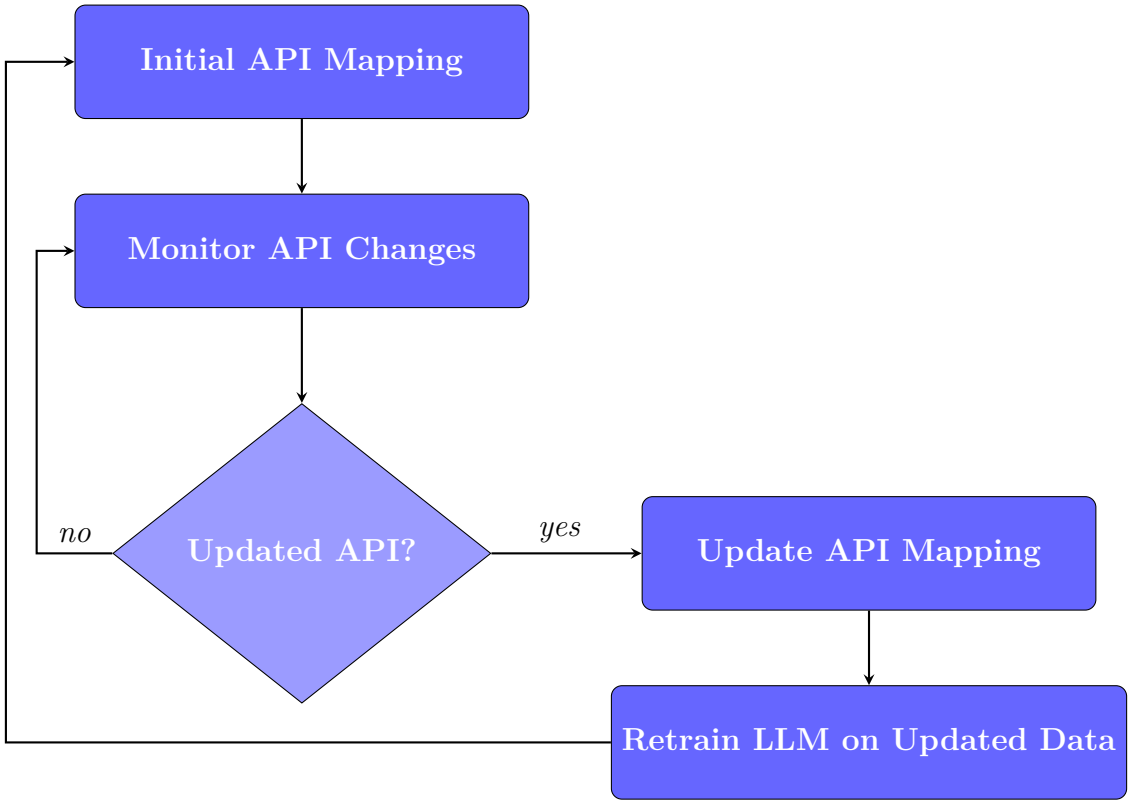

Figure 2.6: Dynamic adaptation process in the SEAL framework.

Chapter 3

Methodology

3.1 Research Roadmap

This research follows a structured roadmap designed to systematically investigate the role of Large Language Models (LLMs) in bridging the gap between abstract requirements and concrete API functionalities. The study is guided by a set of fundamental research questions, which served as the foundation for our methodology and ensured a focused exploration of key challenges and opportunities.

3.1.1 Phases of Investigation

The research follows a two-phase approach, progressively refining techniques and methodologies to maximize the effectiveness of LLMs in Requirements Engineering (RE).

- **Phase 1: Standard Prompting and Initial Analysis**
  The first phase focuses on exploring how LLMs can facilitate the alignment between high-level requirements and API functionalities. In this stage, standard prompting techniques are used without exploiting extensive refinement. This is done to establish a baseline for the ability of the model to map requirements to APIs. The aim is to evaluate the initial performance of LLMs in handling requirement elicitation and identifying their strengths and weaknesses in requirement mapping.
- **Phase 2: Prompt Refinement and Enhanced Mapping**
  Building upon the insights from the first phase, the second phase introduces advanced prompt engineering techniques to refine and enhance requirement mapping. This phase investigates how various refinements, including Chain-of-Thought (CoT) prompting, Few-Shot learning, and structured decomposition, can improve the accuracy and consistency of LLM-generated mappings. The research will then focus on iterative improvements, testing different strategies to optimize alignment between requirements and API functionalities.

3.1.2 Core investigation topics

This study aims to address four fundamental research questions, each contributing to a comprehensive understanding of LLMs' role in Requirements Engineering. .

1. **How can LLMs support requirements elicitation, analysis, and specification?**
   Requirements elicitation is often hindered by ambiguity and incomplete stakeholder input. This research explores whether LLMs can analyze textual requirements, infer missing elements, and structure requirements in a way that improves clarity and completeness. Furthermore, the study provides an investigation about the ability of the GPT model to assist in requirements analysis by identifying patterns, inconsistencies, and dependencies across different specifications.

2. **How effective is prompt engineering in improving the mapping accuracy?**
   The study examines whether different prompt engineering techniques influence the ability of LLMs to accurately map requirements to API functionalities. The research compares basic prompting techniques with advanced methodologies such as context-aware prompting, constraint-based prompting, and iterative refinement to determine their impact on mapping precision.

3. **Can LLMs enhance API alignment in Goal-Oriented Requirements Engineering (GORE)?**
   GORE frameworks focus on defining system objectives based on stakeholder goals, but the transition from goals to implementation remains a challenge. This research investigates whether LLMs can assist in breaking down high-level goals into structured subgoals and align them with suitable API calls, thus improving the coherence of the requirements engineering process.

4. **How can Semantic API Alignment (SEAL) improve the translation of high-level goals into actionable API calls?**
   SEAL provides a methodology for linking user-centric goals to API functions, but its effectiveness when combined with LLMs has not been extensively studied. This research explores whether integrating LLMs with SEAL can result in more accurate and context-aware requirement-to-API mappings, potentially improving automation in software development processes.

All of these research questions play a crucial role in determining the viability and reliability of the following proposed approach.

3.2 The proposed approach

To evaluate the feasibility of this approach, a series of **structured** conversational **interactions** are conducted with the GPT-4 engine using the ChatGPT online interface. Tailored prompts are designed to emulate the autonomous behaviors expected of each distinct agent present in the architecture. These interactions focus on a range of applications sourced from the EMB database, maintaining alignment with those chosen in "*Semantic API Alignment: Linking High-level User Goals to APIs.*", Feldt, R., & Coppola, R. (2024) [16], the article used as a crucial reference for this research.

It is crucial to remember that, at this initial stage, the advanced techniques offered by prompt engineering will not be leveraged. As already pointed out, this choice ensures a foundational assessment of the methodology's feasibility based solely on standard prompts, allowing for a clearer evaluation of the baseline capabilities of the conversational agent in mapping goals to API interactions.

3.2.1 Structure of the tailored prompts

The first prompt (**PR1**) is designed to **identify the potential stakeholder** relevant to the application under scope. To achieve this, the stakeholder selection outlined in Coppola's case study, the *Catwatch* application, is used as a reference point.
This prompt aims to align ChatGPT's responses with the stakeholder framework defined in previouss research, effectively "training" the model to mirror the stakeholder identification process within a similar context.

Prompt PR1
I need you to choose a stakeholder based on the reasoning already made on the choice of this one: *__Context__:* You are assisting in the goal elicitation process for: [*CatWatch is a web application that fetches GitHub statistics for your GitHub accounts, processes, and saves your GitHub data in a database, then makes the data available via a REST API. The data reveals the popularity of your open source projects, most active contributors, and other interesting points. As an example, you can see the data at work behind the Zalando Open Source page. To compare it to CoderStats: CatWatch aggregates your statistics over a list of GitHub accounts.*]. *__Stakeholder Description__:* [*Owner of a GitHub account*] So, if the stakeholder chosen for the CatWatch application was the Owner of a Github account, what could be, using the same identical reasoning, the stakeholder for the following application: [*Description of the application in the Database EMB*]

Table 3.1: Structure of prompt PR1.

The second prompt (**PR2**) asks the model to **generate high-level goals** for the application from the perspective of the identified stakeholder.
This step aims to capture the primary objectives and expectations of the previously identified stakeholder (obtained as an output from PR1), establishing a framework of goals that drive the application's purpose. It is of extreme importance to mention that only the goals given from a final user perspective are going to be considered, ignoring the goals given from a development perspective.
Moreover, it is worth mentioning that this mapping was performed through just one iteration. The mapping will then be refined as the research goes on thanks to:

- The application of additional iterations.
- The use of advanced Prompt Engineering techniques.

Prompt PR2

** **Prompt**: "Elicit High-Level Goals for a Specific Stakeholder in a New Software Project"**

__Context__: You are assisting in the goal elicitation process for *:[Description of the application]*

__Stakeholder Description__: *[Output of Prompt PR1]*

__Task__: Based on your understanding of the typical needs and interests of this specific stakeholder in such projects, help generate a list of high-level goals.

For each goal:
1.**Provide a One-Sentence Description:** Clearly state the goal, considering the stakeholder's perspective and needs.
2. **Explain the Motivation:** Provide a one-sentence explanation of why this goal is relevant and important for this specific stakeholder.

Consider aspects such as:
- **Functional Needs**: What are the primary functionalities or services the stakeholder expects from the software?
- **Quality and Performance Expectations**: Are there specific standards or performance criteria the stakeholder might prioritize?
- **Operational or Business Objectives**: How does the stakeholder intend to use the software to achieve their operational or business goals?

Please ensure that the goals are realistic, clearly articulated, and align with the general objectives and challenges typical to the stakeholder's role or interests in such software projects.

Table 3.2: Structure of prompt PR2.

The third prompt (**PR3**) instructs the agent to **break down the high-level goals** identified in the previous step **into** a more detailed set of **low-level goals**.
This decomposition process is designed to outline specific, actionable objectives that align with the broader aims of the stakeholder, facilitating a structured path from overarching goals to concrete application functionalities.

Prompt PR3

Prompt: "Elicit Low-Level Goals for a Specific Stakeholder in a New Software Project"**

***Context**:* You are assisting in the goal refinement process for a software. The high-level goals of the software are the following: [*Output of Prompt PR2*]

***Task**:* Based on your understanding of the typical tasks that compose the sequence of high-level goal, provide, if possible, a decomposition of goals into subgoals. Each low-level goal should theoretically correspond to a single action of the actor with the software.
Please give goals from the perspective of the user using the application, not development goals.

For each goal:
1. **Provide a One-Sentence Description:** Clearly state the goal, considering the stakeholder's perspective and needs.
2. **Explain the Motivation:** Provide a one-sentence explanation of why this goal is relevant and important for this specific stakeholder.

Please ensure that the goals are realistic, clearly articulated, and align with the general objectives and challenges typical to the stakeholder's role or interests in such software projects.
Please also make sure that the goals are goals of the actors using the software, and not development goal for the implementation of the software.

Table 3.3: Structure of prompt PR3.

The fourth prompt (**PR4**) directs the agent to interpret the JSON structure of the API and **generate** a detailed list of available **endpoints**.
For each endpoint, the prompt specifies the inclusion of key details required for usage: the endpoint's name, a brief description, supported HTTP verbs, parameters (including type and name), output format, and the expected result. This structured output enables a clear and actionable summary of the API, supporting efficient integration and use by the agents.

Prompt PR4

Prompt: "Extract API documentation from a Swagger FILE"

***Context**:* You are assisting in the API abstraction of a given software. The swagger file documenting the APIs of the software is the following: [*SWAGGER FILE*]

Task: please list all the endpoints from this swagger file excerpt and list for each of them:
- the path of the endpoint
- the verb used for the endpoint
- the tag of the endpoint
- the summary of the endpoint
- the description of the endpoint
- the operationId of the endpoint
- the consumed and produced type of the endpoint
- the parameters to call the endpoint.

Table 3.4: Structure of prompt PR4.

The fifth prompt (**PR5**) takes as input the list of low-level goals generated from P3 and the details of the REST endpoint from P4. Its purpose is to create a **1-to-many mapping** between these low-level goals and the corresponding API calls.
This mapping serves as a critical step in assessing the feasibility and practicality of the research approach, determining if the specified goals can be effectively addressed through available API interactions.

Prompt PR5
Prompt: "Mapping between API endpoints and low-level goals" ** *Context:* You are assisting in the mapping of goals of a software to its API endpoints. The low-level goals of the software are the following: [*Output of PR3*] The endpoints of the software are the following: [*Output of PR4*] *Task*: please map each goal to a sequence of calls to the given endpoints. If a goal is not mapable to an endpoint, please write that the goal cannot be enforced by using the current set of endpoints.

Table 3.5: Structure of prompt PR5.

3.2.2 How the prompts are interconnected

The output of certain prompts serve as input for subsequent ones, creating a **sequence** of interconnected interactions. Going deeper on the level of details, the process begins with Prompt PR1, which generates an initial output that feeds into Prompt PR2. This output then drives the response of Prompt PR3, continuing the flow of interaction. Notably, Prompt PR5 is influenced by the outputs of both Prompt PR3 and PR4, demonstrating how prompts can branch out and build on multiple prior outputs.

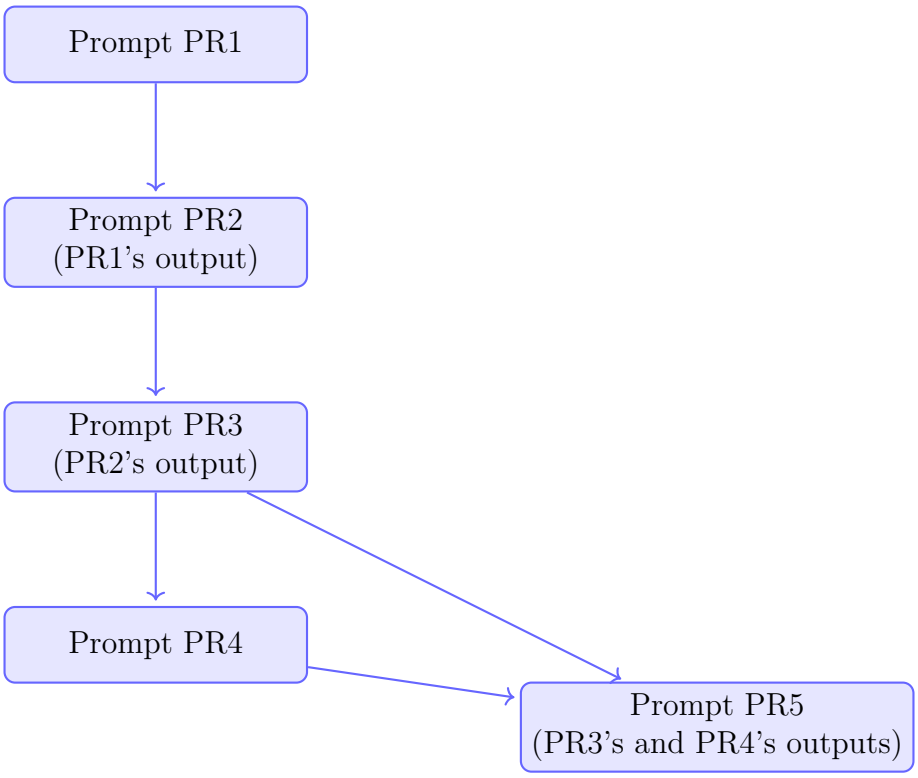

Figure 3.1: How the prompts are interconnected.

The figure illustrates how the output of each prompt shapes the next, ensuring a cohesive and coherent flow of information that refines and evolves the system's responses. This interconnected approach allows for a more contextually aware and responsive process.

3.3 Inspirational example

As a case study, the research started through the testing on an application from the EMB database known as the '*Features Model*.' This example was essential to gain a clearer understanding of the functionality of the methodology and to evaluate its feasibility. Moroever, this application was chosen as the starting point for our research particularly due to its completeness in both API documentation and Github description.

3.3.1 Description of the case study

In the second prompt (PR2), the model is instructed to generate high-level goals based on the application's description. To facilitate this, the GPT interface was provided with a description extracted from the GitHub repository. However, before moving to the testing phase, it is crucial to first understand the nature of the application under scope. A **feature model** is a structured and concise representation of all products within a Software Product Line (SPL), organized around the concept of features.
SPL development emphasizes the systematic and efficient creation of a family of related software products that share common features while accommodating variations based on specific needs. The goal is to offer a REST-based service for:

- **Defining Products and Their Features**.
  This involves specifying the set of products within the SPL, detailing the features available for each product, and outlining the activation constraints that govern the interdependencies and relationships between those features.

- **Defining Product Configurations**.
  A product configuration is defined as a specific combination of active features of the available set. These configurations must adhere to the feature activation constraints, ensuring that the resulting configuration is valid and meets the product's design criteria.

- **Querying Active Features for a Configuration**.
  This functionality allows querying the active features of a configuration. By leveraging this capability, an application can dynamically adjust its behavior during run-time. The behavior may be influenced by various factors, such as the logged-in user, the client, or any other contextual information. This enables a flexible and context-aware execution of the software, optimizing its performance and user experience based on the active configuration. In essence, this service allows for precise control over the set of features activated in a product, ensuring that the software behaves consistently according to predefined constraints while supporting flexibility for different use cases.

This application served as our guiding example, shaping the process all the way from PR1 to PR5.

3.3.2 Testing the case study

Initially, the description of the application under scope in the GitHub database contained limited information. To deal with this, further research was conducted to provide the

agent with a sufficiently detailed description. The following description was provided to the ChatGPT interface in **prompt PR1** and **prompt PR2**.

As defined by Wikipedia, "*a feature model is a compact representation of all the products of the Software Product Line (SPL) in terms of features*".
The focus of SPL development is on the systematic and efficient creation of similar programs.
The goal of this project is to provide a REST-based service for:

- **Defining products**, their available features, and the activation constraints between features.
- **Defining product configurations**, understood as a set of active features of those supported by the product that fulfill the feature constraints.
- **Querying the active features** for a configuration, so an application in run-time can change its behavior according to the active features of a configuration depending, for instance, on the logged user, the client, etc.

The project provides REST resources for defining and querying both products and product configurations.
Let's use as a running example an e-learning website that supplies infrastructure to its clients so they can provide online courses. The company charges its clients according to the enabled features, so the web application adapts its behavior by showing or hiding links and sections when a student is logged in. Some of the supported features are video lessons, online forums and chats for support, payments with credit card, PayPal or wires, redeem codes, etc.

Working with products:

- **Add a product**
- **Add new features**
- **Remove a feature**
- **Request a list** with the names of all available products
- **Request the features and constraints** of a product
- **Remove an existing product** and all its configurations

Table 3.6: Description of the case study: Features Model.

Working with products:
Feature constraints can be removed after adding a product
Automatic feature inclusion: When a feature requires another feature, it will be automatically added to the configuration when the source feature is added.
Feature exclusions: When a feature excludes another feature, it will turn the configuration invalid if both of them are active.
Add a new configuration for one product
Activate/Deactivate a feature in a configuration
Remove a configuration
Query all the active feature names in a configuration

Table 3.7: Description of the case study: Features Model, continuation.

Starting with the initial description and systematically tracing the entire sequence from PR1 to PR5, the first mapping was conducted. It is essential to acknowledge that the mapping was a **1:many** one, between each individual API endpoint, extracted from the SWAGGER file, and its corresponding low-level goal, with the possibility for one individual API endpoint to cover several low-level goals. This step was crucial in validating the alignment between the extracted API functionalities and the intended objectives.

Moreover, it is important to highlight that, as the research progresses, a recurring pattern emerged. Specifically, the model demonstrated a strong ability to accurately map certain categories of goals while consistently omitting others. This observation will be further explored in the second phase to understand the underlying reasons behind these exclusions and their potential implications.

High-level Goal	Low-Level Goals	Mapped Endpoint
Maximize Revenue Opportunities	1.1 Set different pricing tiers for feature bundles **1.2 Track feature usage for billing purposes** 1.3 Enable promotional discounts on features	1.1 Cannot enforce with current set of endpoints. 1.2 `GET /products/{productName} /configurations/{configurationName} /features` 1.3 Cannot enforce with current set of endpoints.
Enhance product customizability	**2.1 Allow clients to customize their feature sets** 2.2 Enable personalization based on user roles **2.3 Track client-specific configurations and feature usage** **2.4 Provide tools for clients to manage their configurations**	2.1 `POST /products/{productName} /configurations/{configurationName}` - Add a configuration to a product. `POST /products/{productName} /configurations/{configurationName} /features/{featureName}` - Add a feature to a specific configuration. 2.2 Cannot enforce with current set of endpoints. 2.3 `GET /products/{productName} /configurations/{configurationName} /features` - Get active features for a configuration. 2.4 `GET /products/{productName} /configurations` - Get configurations for a product. `GET /products/{productName} /configurations/{configurationName}` - Get configuration by name for a product. `PUT /products/{productName} /configurations/{configurationName}` - Update a specific configuration (implied).
Improve User Experience	3.1 Provide a user-friendly interface for configuration management 3.2 Generate detailed analytics and reports for clients	Cannot enforce with current set of endpoints. Cannot enforce with current set of endpoints.
Facilitate Efficient Client Management	**4.1 Track client-specific configurations and feature usage** **4.2 Provide tools for clients to manage their configurations**	4.1 `GET /products/{productName} /configurations/{configurationName} /features` - Get active features for a configuration. 4.2 `GET /products/{productName} /configurations` - Get configurations for a product. `GET /products/{productName} /configurations/{configurationName}` - Get configuration by name for a product. `POST /products/{productName} /configurations/{configurationName}` - Add configuration to a product.
Support Continuous Improvement with new features	**5.1 Regularly update the system with new features** 5.2 Conduct periodic reviews and updates based on market trends	5.1 `POST /products/{productName} /features/{featureName}` - Add a feature to a product. `PUT /products/{productName} /features/{featureName}` - Update a feature of a product. 5.2 Cannot enforce with current set of endpoints.

Table 3.8: Features Model: goals and endpoints.

Application	N° of Endpoints	N° of Low-Level Goals	% Low-Level Goals Mapped
Features Model	12	13	**58.33%**

Table 3.9: Statistics about our inspiration example.

The application of Large Language Models in aligning high-level user goals to APIs through Semantic API alignment has demonstrated promising results, with **more than 50%** of low-level goals correctly mapped.
However, despite its potential, this initial approach, which lacks prompt refinement and does not leverage the advantages of prompt engineering and other techniques, has notable limitations that significantly affect its scalability and reliability. The research continued by applying the same approach and methodology to other applications from the same database of *Features Model*, the application presented above.

3.4 Output statistics

This section presents the output statistics obtained from the analysis of APIs using the SEAL framework. A total of **11 applications** from the EMB database were tested, including:

- **Cyclotron**, a web-based platform designed to create and manage dashboards with ease. It provides a built-in editor and a set of customizable components, enabling users, even those without programming skills, to design and modify dashboards effortlessly. Dashboards are structured as JSON documents, which define all necessary properties for rendering, allowing for a streamlined and flexible approach to data visualization.
- **Features Model**, the case study presented above.
- **Language Tool**, an open-source proofreading software that supports over 20 languages, including English, Spanish, French, German, Portuguese, Polish, and Dutch. Unlike basic spell checkers, it detects a wide range of grammatical, stylistic, and contextual errors, helping users improve the accuracy and clarity of their writing.
- **OCVN**, which stands for Open Contracting Vietnam. OCVN is a project designed to streamline the processing and analysis of Vietnam's public procurement data. It enables the import of procurement records from MS Excel into a NoSQL storage system following the Open Contracting Data Standard (OCDS). With built-in visualization tools, OCVN allows users to generate live dashboards featuring charts, maps, and data tables, as well as custom comparison charts. Since the data is stored natively in the OCDS format, it can be efficiently exported without additional information, ensuring high performance and seamless data interoperability.
- **Market**, an e-commerce web application built to facilitate online shopping and order management. It provides a visual representation of products, allowing customers to browse, add, remove, and modify items in their shopping cart before placing an order. The system securely stores registered users' cart contents in a database for future access. On the administrative side, the Market control panel enables efficient product and category management, order tracking, and stock availability adjustments. Orders can be processed from "in progress" to "executed." The platform ensures security through customer registration, authentication, restricted admin access, and double-checking form inputs on both the client and server sides.
- **Bibliothek**, a Java library designed to simplify and optimize file download management within applications. It provides an intuitive API that allows developers to efficiently handle downloads, including starting, pausing, resuming, and canceling them as needed. Key features include progress tracking through listeners, robust error handling, and thread management to ensure downloads do not interfere with the main application process. With its flexible and developer-friendly design, Bibliothek is a powerful tool for integrating seamless and reliable download functionality into Java-based projects.
- **Corona-Warn**, Germany's official contact tracing application designed to help mitigate the spread of COVID-19 using Apple and Google's exposure notification API. Available for both iOS and Android, the app employs Bluetooth technology to exchange anonymous, encrypted data between nearby devices running the app. All

data is stored locally on users' phones, ensuring privacy and preventing unauthorized access.A key component of the system is the Verification Service, which ensures the integrity of upload requests. It includes a Verification Server for request validation, a Verification Portal for hotline employees to generate teleTANs for diagnostic key uploads, an Identity and Access Management system to control access for authorized health personnel, and a Test Result Server to securely deliver lab results to users. This infrastructure enables a secure and privacy-preserving approach to digital contact tracing.

- **Genome Nexus**, a comprehensive platform designed for the rapid, automated annotation and interpretation of genetic variants in cancer. It serves as a centralized resource that integrates data from multiple sources to streamline the analysis of genetic mutations. By converting DNA changes into their corresponding protein alterations, predicting functional effects, and providing insights into mutation frequencies, gene functions, and clinical relevance, Genome Nexus supports high-throughput cancer genomics research. This tool is essential for researchers and clinicians seeking to understand the implications of genetic variations in oncology.

- **Gestao Hospitalar**, a project that aims to develop a tool to support the Sistema Único de Saúde (SUS), Brazil's public health system, one of the largest and most complex in the world. SUS provides universal, free healthcare services ranging from routine checkups to advanced procedures like organ transplants. By ensuring comprehensive health coverage for all citizens, it prioritizes not only treatment but also prevention and quality of life. The proposed tool seeks to enhance SUS efficiency by reducing waste and optimizing resource allocation, starting from patient needs, ultimately improving the system's sustainability and effectiveness.

- **SpaceX**, designed to support the management of space-related data, particularly for capsules and cores, on a data aggregation platform.

- **ReservationsAPI**, which aims to streamline the reservation process, through a simplified reservation management, allowing users to easily book and manage their reservations. The system features user registration and authentication for secure access, ensuring a smooth user-reservation relationship management. It also integrates with existing systems, providing a unified experience. Additionally, it offers comprehensive reporting and analytics to track and analyze reservation data, while ensuring customization and flexibility to meet different user needs and preferences.

Leveraging the same reasoning used for the choice of the *Features Model* application, the 10 remaining applications were selected from the EMB database because they had the most comprehensive and complete API documentation and Github descriptions among all the others.

The data collected include the number of endpoints, the number of low-level goals, and the percentage of low-level goals correctly mapped by endpoints for each application.

Application	N° endpoints	N° low-level goals	% llgoals mapped
Cyclotron	53	35	**17.1%**
Features Model	12	30	**43.33%**
Language Tool	99+	18	**44.44%**
OCVN	99+	17	**33.3%**
Market	13	25	**48%**
Bibliothek	8	20	**5%**
Corona-Warn	5	39	**10.53%**
Genome Nexus	58	34	**85%**
Gestao Hospitalar	12	24	**50%**
SpaceX	20	18	**50%**
ReservationsAPI	7	13	**53.85%**

Table 3.10: Statistics on 11 applications sourced from the EMB database.

The results shown in Figures 3.2 and 3.3 provide key insights about the overall effectiveness of the SEAL framework in mapping low-level goals to API endpoints.

As shown in Figure 3.2, **72.7%** of the applications analyzed achieved a mapping percentage above 30%. Although this result is promising, it may still be insufficient to ensure a reliable and systematic alignment of requirements with API functionalities. In requirements engineering (RE), achieving a higher mapping percentage is crucial to reduce ambiguity and improve software traceability.

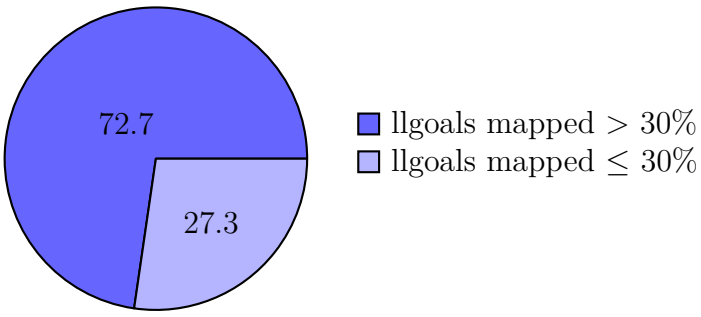

Figure 3.2: Percentage of applications with % llgoals mapped > 30%.

Figure 3.3 further refines this perspective, showing that only **18.2%** of the applications achieved a mapping percentage exceeding 50%. This is particularly relevant because 50% could be considered a minimum threshold for acceptable alignment in RE contexts. A percentage of mapping below this value may indicate difficulties in linking abstract user goals to concrete API functionalities, reducing the practical applicability of the framework in real-world scenarios.

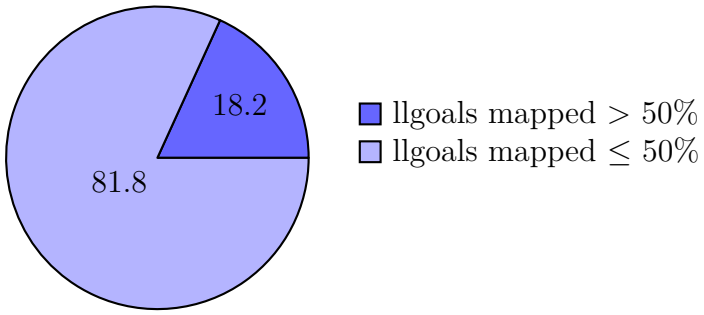

Figure 3.3: Percentage of applications with % llgoals mapped > 50%.

Another important observation is the potential **correlation** between mapping percentages and the comprehensiveness of API documentation and application descriptions. Specifically, applications with long and detailed Swagger files, or those with extensive and structured descriptions on platforms such as GitHub, may exhibit higher mapping percentages. This observation aligns with the one that applications like Genome Nexus (85%) and ReservationsAPI (53.85%), which feature well-documented APIs and descriptions, outperform others like Bibliothek (5%) and Corona-Warn (10.53%), where API documentation might be incomplete or insufficiently detailed.
This suggests that **improving the length and specificity of API descriptions** could be a key factor in improving the performance of the SEAL framework.

3.5 Current Limitations

Despite this initial approach demonstrates high potential in leveraging ChatGPT for requirements engineering (RE) tasks, several limitations impact its effectiveness. These constraints arise from dependencies on contextual clarity, challenges in goal mapping, technical constraints, and broader platform-level limitations. This section outlines the key challenges observed during implementation, highlighting areas where improvements are needed.

1. Contextual and Operational Limitations

A major limitation of this approach comes from its reliance on a comprehensive input context. ChatGPT response quality is highly dependent on the precision and clarity of the prompts. When provided with incomplete or ambiguous inputs, the model often generates inconsistent or irrelevant responses.
Although iterative prompt refinement can mitigate this issue in controlled environments, it becomes impractical in complex RE tasks where obtaining exhaustive context is neither feasible nor efficient.
Additionally, the platform struggles with retaining information across extended multi-turn interactions. This shortcoming hinders its effectiveness in tasks that require persistent contextual memory, such as the hierarchical goal decomposition central to SEAL. In practice, frequent prompt recalibration was necessary to align user goals with API calls, introducing inefficiencies that reduce the scalability of the system, particularly in environments involving multiple stakeholders and intricate software ecosystems.

2. Inconsistencies in Goal Mapping and Output Generation

Another critical challenge lies in the ability of the model to translate high-level goals into actionable steps. Although the ChatGPT interface can generate plausible mappings, it often fails to justify its reasoning or provide transparent explanations for omitted goals. This lack of interpretability introduces barriers to trust and usability, particularly in software engineering contexts where precise traceability and validation are crucial.
It is of crucial importance to note that during the **Goal Elicitation phase**, a notable issue was observed: ChatGPT consistently generates four low-level goals for each high-level goal, regardless of the complexity of the latter. Although this structured approach may seem beneficial, it often results in redundant or irrelevant goals, increasing complexity, and diminishing mapping quality. Some high-level goals inherently require fewer steps, yet the model imposes an arbitrary quota, leading to unnecessary elaboration.
Furthermore, handling **non-functional requirements** remains a significant challenge. Goals related to security, data privacy, and compliance often fail to align effectively with API calls due to their abstract nature and the model's limited understanding of such dimensions. Addressing this issue may require advanced reasoning techniques, such as chain-of-thought (CoT) prompting, which has shown promise in improving logical coherence. Several studies have demonstrated the effectiveness of Chain-of-Thought prompting in enhancing the reasoning capabilities of Large Language Models. For example, Wei in a study conducted during 2022, showed that generating a series of

intermediate reasoning steps significantly improves LLM performance on complex tasks, such as arithmetic and common-sense reasoning. [22].
Additionally, Madaan et al. (2023) found that CoT prompting helps models fill in missing common-sense information, particularly aiding in difficult reasoning problems and long-tail questions. [23]

3. Technical and Architectural Constraints

From a technical standpoint, the reliance of the model on predefined API documentation, such as Swagger files, exposes further limitations. The model struggles to infer or supplement missing details when documentation lacks granularity. For example, if endpoint descriptions do not specify input-output formats, the model may generate incomplete or invalid API call sequences. The absence of self-correction mechanisms exacerbates this issue, as the model does not inherently refine its outputs without external intervention.
Another notable constraint is its sensitivity to **application size and complexity**. As already described previously, the analysis indicates that applications with detailed textual descriptions and fewer endpoints yield better mapping accuracy. The additional context enables more precise goal-to-API alignments, while a lower number of endpoints reduces ambiguity. In contrast, applications with sparse descriptions or a large number of endpoints exhibit reduced mapping precision, as the model struggles to distribute abstract goals across an extensive API landscape.

4. Broader Platform-Level Limitations

Beyond task-specific challenges, ChatGPT and similar language models exhibit broader limitations that impact their applicability in specialized domains. One significant issue is **hallucination**, in which the model confidently generates incorrect or fabricated information. In high-stakes software engineering contexts, this behavior poses substantial risks. Although techniques such as Generate Knowledge Prompting have been explored to mitigate inaccuracies, their practical implementation remains in early stages and requires further refinement.
In this scenario, another systemic challenge is the difficulty of the model in balancing generalization with domain-specific expertise. Although fine-tuning on specialized datasets or leveraging structured prompt engineering methods, such as perspective prompting or the RGC (Role, Goal, Context, Constraint) framework ,could enhance adaptability, these strategies require substantial resources and may not be universally viable.

Overall, these limitations highlight the need for improved contextual understanding, reasoning capabilities, and adaptive methodologies to enhance the effectiveness of the interface in requirements engineering and related domains.

Part II

Core analysis

Chapter 4

Advancing Semantic API Alignment

4.1 A deep dive into Prompt refinement and Prompt Engineering

Prompt Engineering is the process of designing and refining the input prompts provided to Large Language Models, with the aim of maximizing the relevance, accuracy, and utility of their outputs. It operates at the intersection of human creativity and computational reasoning, where the careful crafting of input instructions enables these models to better interpret and generate contextually appropriate responses. [24]
Unlike traditional programming, where algorithms are explicitly coded, Prompt Engineering leverages the pre-trained knowledge embedded in LLMs, using prompts to extract insights or perform specific tasks. This approach transforms vague or complex objectives into structured and actionable instructions that effectively lead the model to the desired outcomes. By doing so, it reduces the ambiguity and enhances the logical consistency of the results generated.

4.1.1 Prompt engineering applicability in the SEAL framework

In the context of our research, Prompt Engineering emerges as a key tool to **bridge** high-level user objectives with the technical functionality of APIs. This connection, facilitated through structured prompts, allows the model to map abstract goals into concrete actions that align with the requirements of stakeholders. The ability to tailor the model responses through carefully constructed prompts ensures that the output remains relevant to the task at hand, minimizing extraneous or irrelevant data. Moreover, it enables iterative refinements, where initial outputs can be evaluated and prompts adjusted to address limitations or inconsistencies, further enhancing the precision of the results.
In a domain like Semantic API Alignment, where the focus is on aligning user goals with API endpoints, Prompt Engineering offers a scalable and flexible solution to manage the complexities inherent in requirements elicitation and mapping. It provides a framework for systematically exploring different perspectives, enhancing outputs, and ensuring that the alignment process is efficient and accurate. This adaptability makes it particularly valuable for addressing the diverse challenges that arise in requirements engineering, from

eliciting stakeholder needs to operationalizing these into technical specifications.

It is crucial to point out that Prompt Engineering is not merely a method of extracting output. Instead, it can be considered as a lens through which complex systems can be understood and optimized. Its iterative nature aligns closely with the principles of our research, which require repeated interactions and refinements to achieve optimal mappings between goals and APIs.
By embedding techniques such as Chain-of-Thought reasoning, Perspective Prompting, and Priming within the prompts, the process facilitates a deeper exploration of the problem space, ensuring that no critical aspect is overlooked. [7]

4.1.2 Key study on the potential impact of prompt engineering

The impact of Prompt Engineering has been extensively studied in recent years. *Google Research* conducted one of the most significant studies in 2022, where **Chain-of-Thought** (COT) prompting, one of the most used and promising techniques of prompt engineering, was evaluated in **PaLM**, a 540 billion-parameter language model. [22]

The research on Chain-of-Thought (CoT) prompting explores how it enhances the performance of Large Language Models (LLMs) by structuring complex reasoning tasks into smaller, intermediate steps. This method is inspired by human cognitive processes, where we break down problems into manageable parts to improve clarity and decision-making. By incorporating these intermediate steps, CoT prompting guides the model through each stage of the reasoning process, allowing it to arrive at more accurate conclusions, especially in tasks requiring multiple steps of logic.

In particular, the study focused on mathematical problem-solving, which often involves intricate multi-step reasoning. To assess the effectiveness of CoT prompting, the researchers used the GSM8K benchmark, a dataset specifically designed to test mathematical reasoning in models. In this case, **accuracy** refers to the model's ability to correctly solve mathematical problems within the benchmark, where each problem requires logical thinking and the ability to break tasks dwon into intermediate steps.
The results revealed the following.

- **PaLM without CoT prompting**
  The model achieved an accuracy of 60%, which means that it correctly solved 60% of the problems but struggled with the multi-step reasoning in the remaining 40%.

- **PaLM with CoT prompting**
  When CoT prompting was applied, accuracy increased significantly up to 80%. This improvement demonstrated the effectiveness of guiding the model through structured reasoning steps, allowing it to solve more problems accurately.

- **Fine-tuned models**
  These models had been specifically trained on the dataset to handle mathematical reasoning and also achieved an 80% accuracy. This result suggested that CoT prompting alone could achieve comparable results to models that were fine-tuned for the task, showing that structured reasoning through prompts could be just as effective as specialized training.
  Summing up, the study shows that prompt engineering techniques like CoT prompting can deliver results that are competitive with more traditional, fine-tuned models,

making it a promising approach for enhancing model performance across a variety of domains. [22]

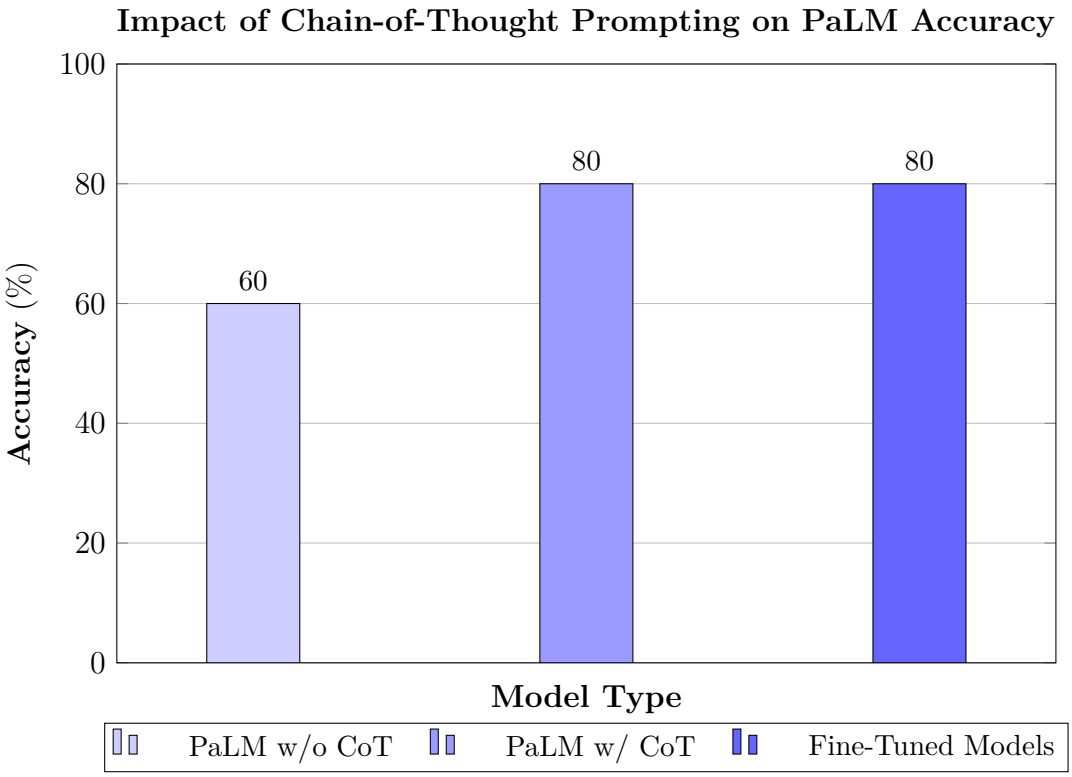

Figure 4.1: Comparison of PaLM accuracy with and without CoT prompting.

4.2 Application of PE in the case study

4.2.1 Techniques employed in the early stage

During the already-shown early phases of this research, prompt engineering was pivotal in tasks such as eliciting user goals, breaking them down into manageable subgoals, and mapping these to API functionalities. To achieve this, the research relied on four key techniques: Priming, Perspective Prompting, Chain-of-Thought (CoT), and the Role-Goal-Context (RGC) framework. These methods were fundamental in the analysis, since they allowed us to tailor the model's responses more precisely, ensuring that the outputs align with the scope of the research.

Priming involves setting the stage for the model by providing it with a detailed introduction to the task at hand. This might include a description of the context, expected outcomes, and even examples to guide the model's understanding. Essentially, priming gives the model a "mental framework" to work within, making its responses more targeted and relevant.

Example of Priming

Task: Generate high-level goals for a software development project.

Context: The goal is to design a web application for an e-commerce platform that allows users to browse and purchase products.

Expected Outcome: High-level goals that focus on enhancing user experience, improving product browsing, and increasing sales.

Application of Priming technique: "You are tasked with generating high-level goals for the development of a web application for an e-commerce platform. The primary focus is on improving the user experience and maximizing sales."

Table 4.1: Application of priming in PR2.

The **prompt PR2**, mentioned above, is a great example of how the **priming technique** is applied effectively. First, it sets the scene by offering a clear **context**. The description of the application and the stakeholder, combined with the purpose of the task, helps the model understand what it is working on and why. This background information is essential to steer the model's focus in the right direction.
Moreover, the task is clearly **defined and broken down**. The model is not simply asked to generate high-level goals; but it is also given instructions on **how to approach the task**. For each goal, it is told to write a one-sentence description, explain its relevance, and consider specific aspects such as functionality, quality expectations, and business objectives. This level of detail reduces ambiguity and ensures that the outputs align closely with the needs of the stakeholder.

Perspective prompting takes the process a step further by asking the model to analyze a task from different points of view. This technique is particularly useful in requirements engineering, where the expectations of various stakeholders, such as developers, project managers, and end users, may differ significantly.

Example of Perspective Prompting

Task: Define requirements for a mobile application.

Developer Perspective: The app must be scalable, fast, and compatible with various devices.

End-User Perspective: The app should be intuitive, easy to navigate, and visually appealing.

Application of Perspective prompting: "From the perspective of the developer, generate high-level requirements for a mobile app, ensuring scalability and performance. From the perspective of the end user, consider usability and design."

Table 4.2: Application of perspective prompting in PR2.

The **prompt PR2** asks the model to generate high-level goals explicitly from the stakeholder's perspective, ensuring relevance to their needs and interests. The emphasis on discarding development-oriented goals further aligns with this technique, ensuring that the generated outputs are tailored to a specific viewpoint.

Another technique is the one mentioned in the last chapter: **Chain-of-Thought** prompting. As already stated, CoT prompting encourages the model to break down complex tasks into small logical steps. By prompting the model to reason through each step of the process, this technique makes its thought process more transparent and easier to evaluate.

Example of Chain-of-Thought

Task: Break down the process of developing a new feature for a mobile app.

Step 1: Identify the feature's purpose and goals.
Step 2: Define the technical requirements and design specifications.
Step 3: Develop the feature and conduct testing.

Application of Chain-of-Thought: "First, clarify the purpose of the feature. Then, outline the technical specifications needed to implement it. Finally, explain how the feature will be tested and validated."

Table 4.3: Application of CoT prompting in PR3.

In **prompt PR3**, the model is asked to decompose high-level goals into low-level, actionable goals. The structure encourages step-by-step reasoning to ensure logical consistency and clear alignment between broader goals and more specific subgoals. Additionally, the prompt's request to explain the motivation for each sub-goal encourages a coherent process.

Finally, the **Role-Goal-Context** framework provides a structured approach to prompt engineering by explicitly defining the model's role, the desired goal, and the specific context of the task. This clarity helps the model stay focused, so as to produce outputs that are highly relevant and actionable.

Example of Role-Goal-Context

Role: Assistant

Goal: Generate high-level requirements for a new software product.

Context: The product is a mobile application designed for online learning.

Application of Role-Goal-Context: "You are assisting in the goal elicitation process for a mobile app designed for online learning. Generate high-level goals that focus on improving accessibility, user engagement, and educational effectiveness."

Table 4.4: Application of Role-Goal-Context, used in all prompts.

The structure of **all the prompts** explicitly defines the role (e.g., "*You are assisting in the goal elicitation process*"), the goal (e.g., "*Generate high-level goals*"), and the context (e.g., "*Description of the application*" or "*Stakeholder description*"). This clear framework helps the model remain focused on the task and produce outputs relevant to the specified role and context.

These techniques were pivotal in shaping the effectiveness and precision of the approach, serving as the foundation to align the model's output with the research objectives and ensuring the robustness of the methodology.

4.2.2 Iterative refinement through new techniques

In the subsequent analysis, new techniques were applied in order to further refine the prompts, addressing the limitations identified in the initial phase and mentioned above, and enhancing the precision and utility of the model's output.
These advanced techniques build upon the foundation established earlier, ensuring that the prompts not only generate relevant results but also do so with greater consistency and alignment to the objectives.

The first technique, **Few-Shot Prompting**, introduces examples of input-output pairs within the prompt to guide the model.
By demonstrating a clear structure and expected outcome, this technique helps the model in understanding patterns and reducing ambiguity in its responses, allowing to align a much higher percentage of low-level goals. In concise words, few-shot prompting leverages the model's pre-trained capabilities by showing it how to approach similar tasks.
For example, in the original task of decomposing high-level goals into low-level ones, the prompt is enhanced to include a sample output. In particular, instead of only stating:

Excerpt from prompt PR3
Decompose the following high-level goals into low-level ones.

Table 4.5: Original prompt PR3.

↓

Prompt refined through the use of Few-Shot Prompting
Decompose the following high-level goals into low-level ones. For example, if the high-level goal is 'Monitor project popularity,' the subgoals could be: 1. Track popularity trends over time; 2. Compare popularity across projects.

Table 4.6: Refined prompt PR3.

The second technique employed, **Fill-in-the-Blank Prompting**, introduces placeholders within the prompt to direct the model toward generating structured and precise outputs. This approach narrows the scope of the task by focusing on specific elements that need completion, effectively guiding the model's attention. For instance, in mapping goals to API endpoints, the original prompt might state:

Excerpt from prompt PR4
List all API endpoints for achieving the goal.

Table 4.7: Original prompt PR4.

↓

Prompt refined through the use of Fill-in-the-Blank Prompting

List all API endpoints for achieving the goal.
Complete the following structure for each API endpoint:

- Endpoint Path: [ ];
- HTTP Verb: [ ];
- Description: [ ].

Table 4.8: Refined prompt PR4.

The final technique, **Self-Critique Mechanism**, prompts the model to evaluate and refine its own output by identifying potential errors or inconsistencies. This iterative approach capitalizes on the model's reasoning capabilities, allowing it to self-correct and improve.

This added layer of introspection ensures that the outputs are not only relevant but also optimized for clarity and alignment.

Excerpt from prompt PR3
Decompose the following high-level goals into low-level ones.

Table 4.9: Original prompt PR3.

↓

Prompt refined through the use of Fill-in-the-Blank Prompting
Decompose the following high-level goals into low-level ones. For example, if the high-level goal is 'Monitor project popularity,' the subgoals could be: 1. Track popularity trends over time; 2. Compare popularity across projects. After listing the subgoals, critique your response by identifying if any subgoals are redundant, unclear, or irrelevant to the high-level goal. Suggest improvements if necessary.

Table 4.10: Refined prompt PR3, after the deployment of 2 techniques.

These new techniques significantly enhance the prompts by introducing examples, structure, and a self-evaluation process. Few-Shot Prompting provides the model with tangible guidance, Fill-in-the-Blank ensures consistent formatting and focus, and Self-Critique Mechanisms elevate the quality of outputs by incorporating an iterative feedback loop. Together, these refinements enable a deeper alignment of the model's outputs with the research objectives, moving closer to achieving robust and actionable results.

4.3 Improving the worst outputs

The application of advanced prompt engineering techniques introduced in the second phase led to a **marked improvement** in the overall performance of the model. By refining the prompts and incorporating strategies such as Few-Shot Prompting, Fill-in-the-Blank Prompting, and Self-Critique Mechanisms, the framework addressed many of the issues identified in the initial phase. These techniques not only enhanced the accuracy and relevance of the outputs but also demonstrated significant potential in resolving some of the more complex challenges inherent in aligning user goals with API functionalities.

To assess the impact of these refined prompts, they were applied to the three applications that had exhibited the poorest mappings during the first phase of the study. These applications had initially suffered from a combination of vague goal decomposition, inconsistent mapping to API endpoints, and redundant output. The results before the second stage of the analysis were the following:

System	N° of Endpoints	N° of LLGs	LLGs Mapped (%)
Cyclotron	53	35	**17.1%**
Bibliothek	8	20	**5%**
Corona-Warn	5	39	**10.53%**

Table 4.11: Summary of outputs without Prompt Engineering.

After implementing the improved prompts, the results showed a substantial enhancement in the quality of the mappings, with clearer goal definitions and more precise alignment to API endpoints. The use of structured templates and iterative feedback mechanisms ensured that previously ambiguous or incomplete mappings were replaced with actionable and coherent results.

System	N° of Endpoints	N° of LLGs	LLGs Mapped (%)
Cyclotron	53	19	**36%**
Bibliothek	8	9	**55.6%**
Corona-Warn	5	10	**60%**

Table 4.12: Summary of outputs with Prompt Engineering.

This notable improvement in goal-to-API mapping quality can be attributed in part to the **decrease in the number of low-level goals** generated for each high-level goal. This reduction in the number of subgoals was a direct result of the refined prompt engineering techniques employed in this phase. Techniques like Fill-in-the-Blank and Few-Shot Prompting helped guide the model toward generating more concise and focused outputs by clearly defining the expected structure and scope of the subgoals. Additionally, the

introduction of Self-Critique Mechanisms encouraged the model to evaluate its responses, eliminating redundant or irrelevant subgoals and focusing only on those with clear and actionable relevance to the high-level objective.

This reduction in subgoals also aligns with the overall goal of minimizing the extraneous cognitive load on stakeholders, ensuring that the generated output remains manageable and directly relevant to the intended application. The streamlined approach highlights how prompt engineering not only improves output quality, but also fosters a more targeted and efficient process to map high-level goals to actionable technical implementations.

4.4 Remaining Limitations and Future Challenges

Despite the significant improvements introduced through the prompt engineering techniques discussed above, several limitations and challenges persist in applying Large Language Models (LLMs) for Semantic API Alignment.
These challenges highlight areas where further refinements and complementary strategies are required to enhance the robustness and reliability of this approach.

1. **Context Retention and Scalability Issues**
   Although techniques such as **Few-Shot Prompting** and **Chain-of-Thought** reasoning have improved the model's ability to generate structured outputs, LLMs still struggle with **context retention** over extended interactions.
   This limitation becomes particularly evident in large-scale projects where multiple iterations are required to refine the goal-to-API mapping. The inability to persistently maintain long-term context necessitates repeated re-feeding of information, leading to inefficiencies in complex requirements engineering scenarios.

2. **Handling of Non-Functional Requirements (NFRs)**
   Our approach demonstrates significant progress in aligning functional goals with API endpoints. However, it still lacks an effective mechanism for handling **Non-Functional Requirements (NFRs)**.
   Security, performance, compliance, and usability-related goals often remain unmapped due to their abstract nature. In this scenario, heuristic approaches and manual interventions can mitigate some of these gaps. However, a more structured framework, possibly integrating domain-specific constraints or external validation tools, is still needed to address NFRs comprehensively.

3. **Dependence on API Documentation Quality**
   The effectiveness of SEAL remains highly dependent on the quality and completeness of available API documentation. If documentation lacks detailed descriptions, parameter specifications, or response formats, the model struggles to make an accurate mapping.
   Automated documentation augmentation techniques or hybrid approaches that combine LLMs with knowledge graphs could be explored to mitigate this dependency.

4. **Handling Edge Cases and Rare Scenarios**
   Current refinements have optimized the model's capability to generalize across common requirements, but the so-called **edge cases** and highly specific scenarios still pose significant challenges.
   Uncommon API behaviors, undocumented features, or context-sensitive implementations often lead to misalignments. Future work should focus on incorporating adaptive learning mechanisms or reinforcement learning techniques to improve LLMs' handling of rare and ambiguous cases.

5. **Need for Continuous Human Supervision**
   Although prompt engineering has enhanced automation in goal-to-API alignment, human intervention remains crucial in verifying mappings, refining outputs, and ensuring accuracy. The model still lacks the ability to self-correct when encountering conflicting or ambiguous requirements.
   Incorporating interactive human-in-the-loop (HITL) frameworks could help balance automation with expert oversight, ensuring higher reliability in real-world applications.

6. **Limited Support for API Evolution**
   APIs evolve over time, with new endpoints being introduced and deprecated regularly. Although some level of dynamic adaptation is achievable through periodic retraining, the current methodology does not offer real-time adaptation to API updates. Future research should explore mechanisms that enable continuous learning from API change logs, version histories, and automated schema analysis to maintain long-term alignment.

Chapter 5

Extending the reach of SEAL

5.1 Categorization of remaining unmapped goals

Despite the advancements introduced through the refinement techniques of prompt engineering, a portion of low-level goals still remains unmapped to API functionalities. Identifying these unmapped goals and understanding their characteristics is crucial to refine the SEAL framework and, in general, the presented approach. The key idea is that if certain categories of goals remain consistently unmapped across all 11 applications, recognizing these recurring patterns could enable targeted model training, improving its ability to detect and address these goals more effectively.

5.1.1 Unmapped categories in our research

By analyzing and categorizing all the unmapped goals across the 11 applications in the EMB database, three main categories emerged as consistently misaligned with API functionalities:

1. **Abstract and Conceptual Goals**
   These goals represent high-level business or strategic objectives that lack a direct, one-to-one correspondence with API operations. Unlike well-defined functional requests, such as retrieving user data or processing a transaction, abstract goals often require a combination of multiple processes, human decision-making, or external analytics to be fulfilled.
   Examples could include *enhance brand reputation, expand market influence, optimize customer engagement strategies.*
   The complexity of these goals stems from their broad scope and the fact that they often involve qualitative rather than quantitative measures, making them inherently difficult to map to discrete API calls.

2. **Non-Functional Goals**
   As already depicted, this category includes goals that pertain to system qualities rather than specific functionalities. These goals define requirements such as to *ensure data security*, *achieve high availability*, or *comply with industry regulations (e.g., GDPR, HIPAA).*
   Since APIs are designed to expose functionalities rather than enforce overarching system properties, non-functional goals often remain unmapped or only partially

addressed through indirect mechanisms. For example, while an API might offer authentication endpoints, ensuring system-wide security involves broader architectural considerations, external monitoring solutions, and compliance audits, which are not inherently covered by API endpoints.

3. **Implicit and Composite Goals**.
   These goals require multiple API interactions, sequential processing, or data aggregation steps to be fully satisfied. Unlike simple API calls that retrieve or modify a single piece of data, implicit and composite goals involve chaining operations across multiple endpoints to derive meaningful results.
   Examples include *generate customer behavior insights from historical transactions*, *create an automated report of system performance metrics*, or *personalize user recommendations based on real-time data*. The failure to map these goals often stems from the absence of built-in orchestration mechanisms that can automatically construct and execute multi-step workflows, forcing developers to implement such logic manually.

5.1.2 Quantification of the unmapped goals

Table 5.1 presents the categorization of unmapped goals based on their characteristics. The distribution highlights the prevalence of abstract and conceptual goals, which constitute the largest portion (45%), followed by non-functional goals (30%) and implicit or composite goals (25%). This classification helps to identify the challenges in aligning certain goal types with concrete implementation steps.

Category	**Percentage of Total Unmapped Goals**
Abstract and Conceptual Goals	45%
Non-Functional Goals	30%
Implicit and Composite Goals	25%

Table 5.1: Categorization of unmapped goals.

It must be acknowledged that failing to map these goals can lead to **several risks**.
When abstract goals are overlooked, the API strategy may diverge from broader business objectives, reducing its overall effectiveness. In the same scenario, ignoring non-functional goals can introduce performance bottlenecks, security vulnerabilities, and compliance issues, making the system more prone to failures. Furthermore, without proper mapping, many composite goals rely on manual intervention, limiting the potential benefits of automation and increasing operational complexity.

5.2 Designing Prompts for Successive Mapping

To address the unmapped categories presented above, this section introduces three distinct **prompting strategies**, each tailored to facilitate the alignment of abstract, non-functional and composite goals with API functionalities. These strategies are grounded in Prompt Engineering techniques, ensuring that the model generates structured and contextually relevant outputs.
It has to be taken into account that this study serves as an initial exploration into the intersection of goal-oriented requirements engineering and LLM-driven API mapping, aiming to establish a methodological foundation for future research. Notably, this approach considers newly formulated general goals rather than those derived directly from the 11 applications sourced from the research, ensuring a broad and adaptable framework for evaluation.

5.2.1 Few-Shot Prompting for Conceptual Goals

Conceptual goals are inherently high-level and abstract, often lacking a direct one-to-one correspondence with API functionalities. Traditional API documentation typically focuses on concrete operations, making it difficult to translate these broad objectives into actionable system behaviors. To cope with this challenge, few-shot prompting provides structured examples within the prompt, guiding the model toward the generation of responses that align conceptual goals with relevant API interactions.

Few-Shot Prompting for Conceptual Goals

Prompt:
Given the goal "*Optimize user engagement*", list API interactions that track user activity, suggest content, and provide recommendations.

Example:

- Retrieve user activity logs via `GET /user/activity`
- Generate personalized recommendations using `POST /recommendations`
- Send engagement notifications via `POST /notifications/send`

Table 5.2: Designing few-shot prompting.

By including explicit examples, the model is guided toward providing structured responses that align with the goal of enhancing engagement, making it easier to link conceptual goals to API actions.

5.2.2 Constraint-Based Prompting for Non-Functional Goals

Non-functional goals, such as security, compliance, and performance, pose a unique challenge in goal-to-API mapping. Unlike functional goals, which directly align with API operations, non-functional requirements typically describe system qualities rather than discrete functionalities. To address this, constraint-based prompting explicitly integrates constraints into the prompt, encouraging the model to evaluate API endpoints based on specific non-functional criteria.

Constraint-Based Prompting for Non-Functional Goals

Prompt: Identify API endpoints that process personal data. For each endpoint, assess compliance with GDPR regulations.

Example:

- `GET /user/profile` - Contains personal data, check encryption policies.
- `POST /user/update` - Modifies user data, ensure access control measures.
- `DELETE /user/remove` - Handles data deletion, verify compliance with retention policies.

Table 5.3: Designing Constraint-based prompting.

By explicitly integrating compliance requirements into the prompt, this approach steers the model toward recognizing quality attributes that influence API selection and usage, making non-functional goal fulfillment more systematic and reliable.

5.2.3 Multi-Step Prompting for Composite Goals

Composite goals involve multiple interdependent actions, requiring a sequence of API interactions to achieve a desired outcome. Unlike single-function mappings, these goals necessitate multi-step reasoning, where each step builds upon the previous one. Multi-step prompting structures the prompt in a way that breaks down composite goals into sequential API calls, enabling the model to generate responses that reflect a logical execution flow.

Multi-Step Prompting for Composite Goals

Prompt: For "*Generate a financial summary*", follow these steps:

Step 1: Retrieve all user transactions using `GET /transactions`
Step 2: Aggregate transaction totals and calculate spending trends.
Step 3: Format results as a structured report using `POST /report/generate`.

Table 5.4: Designing multi-step prompting.

By decomposing high-level objectives into distinct logical steps, this prompting technique enhances the model's planning and reasoning capabilities, ensuring a structured and effective mapping of composite goals to API operations.

By leveraging few-shot, constraint-based, and multi-step prompting, this study improves the alignment of unmapped goals with existing API functionalities, addressing the challenges outlined in previous sections. These strategies serve as a **starting systematic approach** to improve goal decomposition and operationalization, paving the way for more advanced methodologies in LLM-driven requirements engineering and semantic API alignment. Future research can build upon this foundation by refining prompt structures and exploring adaptive techniques that further enhance goal-to-API mappings.

Chapter 6

Conclusions and future perspectives

The integration of Large Language Models (LLMs) in Requirements Engineering (RE), particularly through the framework of Semantic API Alignment (SEAL), has emerged as a **promising approach** for automating the mapping of stakeholders' goals to API functionalities. Although this research has demonstrated significant potential to improve the efficiency of the RE process, it has also highlighted several challenges and limitations that must be addressed for this technology to reach its full potential.

6.1 Key Findings and Research Outcomes

The primary objective of this research was to assess the effectiveness of LLMs in automating the alignment of requirements with API functionalities. The findings indicate that **LLMs can indeed play a crucial role** in facilitating the elicitation process of requirements and mapping abstract goals to actionable API endpoints.
Specifically, the results showed that 72.7% of the applications tested in the first stage of the investigation surpassed a 30% mapping threshold. This indicates that, in the initial stages, LLMs are capable of identifying and linking a significant portion of the required goals with relevant APIs.

In addition, the refinement of the prompting techniques during the second stage of the research led to substantial improvements in mapping precision. The targeted prompt engineering resulted in an increase in mapping accuracy up to 40% for selected applications, and most of all, all 11 applications tested showed a mapping percentage above 30% in this second stage. This suggests that, with iterative refinements, LLMs can progressively improve their ability to align requirements with API functionalities, further improving the automation of the RE process. These findings underscore the promising role of LLMs in enhancing the efficiency of RE, especially when coupled with appropriate prompt engineering techniques.

6.2 Limitations and Challenges

Despite the encouraging results, several challenges and limitations remain. These issues hinder the scalability and reliability of LLM-driven SEAL frameworks.
One of the most significant issues is the inconsistency and **incompleteness** often found **in the API documentation**. Many APIs lack clear, standardized documentation, which complicates the task of aligning abstract requirements with available functionalities. In such cases, LLMs can struggle to identify the correct API endpoints or may generate incomplete or inaccurate mappings, which can undermine the effectiveness of the approach.

Another limitation is the reliance on manual prompt engineering and iterative refinements. Although prompt engineering is crucial to improve the alignment process, it remains a labor-intensive task that requires significant expertise. The **need for** continuous **human intervention** raises concerns about the scalability of this approach. As the complexity of requirements increases, the need for more refined and tailored prompts grows, further increasing the burden on human engineers. This reliance on manual adjustments makes the process less autonomous, reducing the overall efficiency of the system.

Furthermore, the current state of LLMs alone is insufficient to achieve a fully autonomous and reliable solution for requirements engineering. Despite showing significant promise, LLMs are still heavily dependent on the quality of the input prompts and the training data. In particular, LLMs may fail to handle edge cases, rare requirements, or unforeseen complexities in the requirements, which can significantly impact the accuracy of the mapping process.

6.3 The Need for Hybrid Approaches

Given these limitations, it is clear that LLMs alone cannot provide a fully autonomous and scalable solution, due to the complexities inherent in requirements engineering. Future advancements in this area should focus on **integrating hybrid AI approaches**, which should combine the strengths of LLMs with other artificial intelligence techniques, such as rule-based systems, constraint solvers, and expert-driven knowledge bases. [25]
These hybrid approaches can help address the challenges posed by inconsistent API documentation, rare requirements, and complex mappings. Several specific hybrid AI techniques that could be integrated into LLM-driven SEAL frameworks are the following:

- **Integration of Knowledge Graphs**

  Knowledge graphs represent domain-specific information in a structured format, capturing entities and relationships within a given domain. By integrating knowledge graphs with LLMs, the model can leverage the graph's structured data to better understand the context of the requirements and identify relevant API endpoints.
  This integration allows LLMs to map abstract goals to APIs more effectively by understanding the underlying relationships between different components within the system. For example, if a goal is related to processing user data, the knowledge graph can highlight relevant APIs related to data handling and privacy compliance.
  Moreover, knowledge graphs provide a rich source of domain expertise, which can aid the model in handling complex or domain-specific requirements, thereby improving the robustness and relevance of the API mappings. [26]

- **Incorporating Rule-Based Systems**

  Rule-based systems can enhance the decision-making capabilities of LLMs by applying pre-defined rules that guide the mapping process. These rules can encode domain-specific knowledge, such as API usage constraints, regulatory requirements (e.g., GDPR), and performance thresholds.
  For example, a rule-based system could enforce that all APIs related to user authentication must adhere to specific security protocols, such as OAuth2. When integrated with LLMs, such rules can provide a layer of validation to ensure that the generated API mappings meet non-functional requirements. The combination of rule-based systems and LLMs allows for the best of both worlds: LLMs provide flexibility and adaptability in generating responses, while the rule-based systems ensure that these responses are compliant with established standards and requirements. [27]

- **Constraint Solvers for Non-Functional Goals**

  Non-functional requirements (NFRs) such as security, scalability, and performance often require precise constraints to be applied during the API selection process. Constraint solvers are specialized algorithms designed to optimize solutions based on a set of constraints. By integrating constraint solvers with LLMs, the system can be more effective in selecting APIs that not only fulfill functional requirements but also meet non-functional goals.
  For instance, the solver can ensure that the chosen API for data storage complies with data retention policies or that it meets latency thresholds. This hybrid approach improves the accuracy of API mapping, especially when dealing with complex NFRs that require specialized knowledge beyond the capabilities of LLMs alone. [25]

- **Human-in-the-Loop (HITL) Integration**:

  Although LLMs show impressive results, human expertise is still needed to ensure the correctness and alignment of the generated mappings. A Human-in-the-Loop (HITL) approach allows human experts to step into the process, reviewing and refining the LLM-generated API mappings.
  HITL can be particularly useful in cases where the LLM encounters ambiguity or when dealing with rare, complex, or highly specific requirements that the AI model might not fully understand. Humans can provide judgment and domain knowledge to correct or guide the model's output, improving the overall accuracy and reliability of the solution. Moreover, HITL integration facilitates the continuous learning process of the LLM. As humans provide feedback and corrections, the model can improve over time, becoming more adept at handling similar cases in the future. [28]

- **Reinforcement Learning (RL) for Iterative Refinement**:

  Strongly related to the previous one, another important approach to improve the current model is Reinforcement learning (RL). RL is an AI technique where models learn through trial and error by receiving feedback from their actions. Integrating RL with LLMs can allow the system to iteratively refine its mappings by learning from its mistakes.
  In the context of API mapping, RL can be used to continuously improve the model's decision-making process by rewarding correct mappings and penalizing incorrect ones. This approach helps optimize the API selection process, especially when the

requirements are complex or evolve over time.
RL can also be used to adapt the system to new, previously unseen requirements, making the approach more flexible and dynamic in addressing evolving needs. [29]

6.4 Future Perspectives and Research Directions

The promising results coming from this study suggest several avenues for future research. One key direction is the continued development and refinement of **prompt engineering techniques**. Future work could explore the use of more advanced prompt structures, such as few-shot learning and zero-shot learning, to further enhance the ability of LLMs to generalize and apply their knowledge to a broader range of requirements and API interactions.
In addition, further investigation into the integration of **hybrid AI approaches** will be crucial to overcome the limitations of current LLM-based solutions. Research into combining LLMs with other AI techniques, such as machine learning, reinforcement learning, and knowledge-based reasoning, could lead to more robust and scalable solutions for API mapping. Specifically, multi-modal approaches, which combine textual data from LLMs with other forms of structured data (e.g., API documentation, code repositories, and system specifications), could enhance the model's ability to handle complex and diverse requirements.
Another important area of future research involves improving the **evaluation metrics** used to assess the effectiveness of LLM-driven SEAL frameworks. Current evaluation methods, such as accuracy percentages, do not always capture the full complexity of the task. Future research could explore the use of more sophisticated metrics, such as the F1-score, to provide a more nuanced understanding of the feasibility and performance of the model.
In addition, conducting real-world case studies and pilot projects will be essential to assess the practical applicability and scalability of LLM-based solutions in different domains.
Finally, **collaboration with industry** is essential to bridge the gap between academic research and real-world applications. Engaging with industry professional to better understand the challenges they face in requirements engineering and API mapping can help shape the development of more effective and practical solutions.

Bibliography

[1] Sommerville, I. (2016). *Software Engineering.* Addison-Wesley.

[2] Nuseibeh, B., & Easterbrook, S. (2000). *Requirements engineering: a roadmap.* In *Proceedings of the Conference on The Future of Software Engineering* (pp. 35-46).

[3] Zave, P., & Jackson, M. (1997). *Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(1), 1-30.

[4] Kumar, A., et al. (2021). *Enhancing REST API design using natural language processing techniques.* arXiv preprint arXiv:2107.11000.

[5] Hora, A., Robbes, R., Valente, M. T., Anquetil, N., Etien, A., & Ducasse, S. (2018). How do developers react to API evolution? A large-scale empirical study. *Software Quality Journal, 26*(1), 161–191.

[6] Liu, Y., et al. (2020). *T5: Text-to-Text Transfer Transformer.* arXiv preprint arXiv:1910.10683.

[7] Brown, T. B., et al. (2020). *Language models are few-shot learners.* arXiv preprint arXiv:2005.14165.

[8] Misra, S., et al. (2011). *Application of machine learning techniques to software engineering. ACM Computing Surveys (CSUR)*, 43(4), 1-35.

[9] Vaswani, A., et al. (2017). *Attention is all you need. Advances in Neural Information Processing Systems*, 30.

[10] Radford, A., et al. (2019). *Language models are unsupervised multitask learners. OpenAI Blog*, 1(8), 9.

[11] R. K. M. & P. H. A., "Ambiguity in Organizational Goals and its Effect on Performance," *Journal of Public Administration Research and Theory*, vol. 12, no. 3, pp. 345-362, 2000.

[12] Cleland-Huang, J., et al. (2010). *Automated support for managing feature requests in open source software. Communications of the ACM*, 53(10), 109-116.

[13] Kumbhar, A. T., et al. (2021). *Requirements Engineering Meets Artificial Intelligence: A Survey.* arXiv preprint arXiv:2107.11000.

[14] Garcez, A. S., & Zaverucha, G. (1999). *The connectionist inductive learning and logic programming system. Applied Intelligence*, 11(1), 59-77.

[15] Ernst, N. A., et al. (2012). *Automated analysis of requirements evolution using natural language processing.* In *Requirements Engineering Conference (RE), 2012 20th IEEE International.*

[16] Feldt, R., & Coppola, R. (2024). *Semantic API Alignment: Linking High-level User Goals to APIs.* arXiv preprint arXiv:2405.04236.

[17] Settles, B. (2009). *Active learning literature survey.* University of Wisconsin, Madison.

[18] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E., Le, Q. V., & Zhou, D. (2022). *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. Google Research.*

[19] Oo, K. H., Nordin, A., Ismail, A. R., & Sulaiman, S. (2018). An Analysis of Ambiguity Detection Techniques for Software Requirements Specification (SRS). *International Journal of Engineering & Technology, 7*(2.29), 501–505.

[20] Cheng, H., Husen, J. H., Lu, Y., Racharak, T., Yoshioka, N., Ubayashi, N., & Washizaki, H. (2024). Generative AI for Requirements Engineering: A Systematic Literature Review. *arXiv preprint arXiv:2409.06741.*

[21] Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Schulman, J., Hilton, J., Plappert, M., Kosaraju, V., Sastry, G., Mishkin, P., Chan, B., Gray, S., ... Sutskever, I. (2021). Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374.*

[22] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv preprint arXiv:2201.11903.

[23] Madaan, A., Hermann, K., & Yazdanbakhsh, A. (2023). What Makes Chain-of-Thought Prompting Effective? A Counterfactual Study. Findings of the Association for Computational Linguistics: EMNLP 2023, 1448–1535.

[24] Chen, B., Zhang, Z., Langrené, N., & Zhu, S. (2024). Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review. *arXiv:2310.14735.*

[25] Smith, A., & Johnson, R. (2021). Hybrid AI Approaches in Software Engineering: A Comprehensive Review. *Journal of Artificial Intelligence in Engineering*, 18(3), 210-225.

[26] Li, Y., & Wang, M. (2023). Integrating Knowledge Graphs with Large Language Models for API Mapping. *Journal of AI Integration*, 25(1), 45-60.

[27] Zhang, X., & Liu, H. (2022). Rule-Based Systems in Software Development: Benefits and Challenges. *Journal of Systems Engineering*, 32(4), 350-366.

[28] Li, Z., & Zhang, Y. (2023). Human-in-the-Loop approaches for enhancing AI systems in complex decision making. *AI and Systems Engineering Review*, 28(3), 240-255.

[29] Williams, T., & Brown, L. (2022). Reinforcement learning for continuous improvement in AI systems. *Journal of Artificial Intelligence Research*, 50(2), 1.