POLITECNICO DI TORINO

Department of Computer Engineering, Cinema and Mechatronics

Master Thesis

Enhancing Requirements Engineering with Large Language Models: From Elicitation and Classification to Traceability, Ambiguity Management and API Recommendation

Supervisors
Prof. Riccardo COPPOLA

Candidate
Vittoria OCLEPPO

Academic Year 2024-2025

Abstract

This thesis investigates the integration of Large Language Models (LLMs) into Requirements Engineering to enhance the overall management of software requirements. The research develops an extensive literature review that begins with the evolution of sequence modeling—from RNNs to the attention mechanism and encoder-decoder frameworks—and proceeds by examining a large number of the most recent approaches that apply Pre-Trained and Large Language Models to the key requirements engineering tasks—from elicitation and classification to ambiguity management, traceability and finally API recommendation.

Along with this, two practical experiments on API tracing are conducted. The experiments employ a multi-step approach based on Goal-Oriented Requirements Engineering (GORE) to map low-level software goals—generated based on natural language project documentation—to specific API endpoints detailed in Swagger files. The experiments simulate the distinct tasks of LLM-based agents, either through iterative conversational interactions with GPT-4 via ChatGPT or leveraging an implemented code solution that extends the ChatGPT experiment with support for multiple LLM providers and a refinement mechanism for evaluating and improving results. The results highlight that while the model can effectively generate and map user-centric goals to API endpoints, performance is variable, with occasionally inconsistent mappings and omissions, which underscores the need for further interventions aimed at improving results, including enhanced prompt engineering, quality and clarity of software description and, ultimately, more advanced approaches leveraging an agentic architecture.

Overall, this work contributes with an extensive overview of the recent LLM-based approaches to the core tasks of requirements engineering, while also offering practical insights through the experiment part in eliciting software goals and mapping them to API endpoints, showing challenges and benefits of applying LLMs to requirements engineering.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, there has been a growing interest in leveraging Artificial Intelligence (AI) to enhance practices within the Software Engineering (SE) field, which is concerned with the development, implementation, and maintenance of software systems. Specifically, under an innovative perspective, code can be treated as a special form of structured natural language, hence many SE tasks can be reframed as Natural Language Processing (NLP) tasks.

Natural Language Processing, a subfield of AI, combines Computational Linguistics with Machine Learning and Deep Learning techniques to enable software systems to understand and process natural language and accomplish tasks.

This paradigm shift opens up new opportunities for AI to impact critical aspects of the software development lifecycle. One of the most promising applications of NLP in SE is in the domain of Requirements Engineering (RE). RE is a crucial phase in software development that deals with identifying, documenting, and managing software requirements, which form the foundation of any software system.

Properly managing requirements is vital for ensuring that the final system works properly and fully aligns with the needs and expectations of stakeholders. However, the traditionally manual nature of this process makes it challenging, time-consuming and error-prone.

In this context, the integration of AI technologies, particularly Large Language Models (LLMs), has the potential to automate, streamline, and enhance many facets of the requirements engineering process, from requirements elicitation and classification to traceability and completeness analysis, thereby reducing manual effort and increasing the accuracy of the final software specifications. These models, trained on extensive corpora and with powerful understanding capabilities, are well-suited to assist in the various stages of the requirements engineering process. By streamlining these tasks, LLMs can significantly enhance the efficiency and quality of software engineering processes, ultimately leading to more effective software systems and better alignment with stakeholder needs.

The primary objective of this thesis is to investigate how LLMs can be applied to the following key areas of Requirements Engineering:

- Streamlining Requirements Elicitation: leveraging LLMs to automate and optimize the gathering of requirements, which includes generating interview scripts, synthesizing information from diverse sources, and reducing the time required for the initial stages of requirements collection.

- Classification and Categorization of Requirements: using LLMs to help in the classification and categorization of requirements into functional, non-functional, and other relevant categories, which is an important aspect to ensure that the requirements are structured and easily manageable.

- Managing Ambiguity and Enhancing Completeness: applying LLMs to improve the clarity and completeness, which is a critical challenge due to the often ambiguous or incomplete nature of requirements.
- Improving Traceability: leveraging LLMs to automate the creation and management of traceability links, making it easier to track the relationship between requirements, design, implementation, and testing and ensuring all requirements are properly addressed during the software development life cycle.
- Automating API Recommendation: using LLMs to recommend suitable APIs that align with the identified requirements, facilitating the development process by reducing manual effort.

This thesis is structured as follows:

Chapter 2 provides an overview of the existing research and methodologies relevant to this thesis. It introduces the concept of Natural Language Processing and its application to software engineering tasks. The chapter also discusses sequence modeling frameworks and the different LLM architectures, including encoder-decoder, encoder-only, and decoder-only models, which form the theoretical foundation for the works analyzed in later chapters.

Chapter 3 delves into the specific application of LLMs to Requirements Engineering. It includes a comprehensive literature review exploring how LLMs can be used for requirements elicitation, classification, ambiguity management, traceability, and API recommendation. The chapter formulates key research questions and discusses findings from several studies in the field that evaluate the effectiveness of LLMs in these areas.

Chapter 4 builds upon the theoretical background and presents a practical experiment on the application of LLMs, particularly through the ChatGPT interface, for API tracing in the context of software requirements. The experiment framework is applied to selected software projects and the results are analyzed to assess the potential of LLMs in streamlining API recommendations. The chapter also includes results and insights from the experiment and suggestions for prompt improvement.

Chapter 5 further extends the Goal-to-API mapping experiment by presenting a code implementation of the experiment workflow enhanced with support for multiple LLM providers and output refinement. The chapter concludes by presenting future directions for improvement.

Chapter 6 finally summarizes in a concise way the key contributions of the research, and suggests directions for future work in this promising field.

By exploring these areas, this thesis aims to provide a comprehensive understanding of how LLMs can enhance the requirements engineering process, make it more efficient, accurate, and scalable, and contributes to the growing body of knowledge in the field of AI-driven software engineering.

Chapter 2

Background

Code modeling and generation initially leveraged rule-based and statistical methods, however, these approaches came with various limitations [1]. For example,

- Domain-specific language (DSL) guided models and probabilistic grammars could effectively capture dependencies and sequential properties, but were hard to generalize and their learning process was harder to automate in new domains, due to strictly defined rules
- N-gram models could automatically extract features and learn dependencies and rules of the programming languages from source code, but could not capture high-level programming paradigms and long-term dependencies well, due to the combinatorial explosion of terms.
- Similarly, simple neural program models (like fully-connected or log-bilinear models) still require human-designed rules to capture the dependencies and structure of source code, which limits its end-to-end training and generalizability.

Over time, these approaches have been replaced by Deep Learning Models. Deep Learning models indeed were more suitable for such tasks since they proved to be better at fundamental aspects such as automatic feature generation, capturing long-term dependencies as well as sequential properties, end-to-end learning, and generalizability.

As said in the Introduction, code can be seen as a particular form of natural language -with its own syntactic structures and semantic meaning-. In this context, several code-related tasks can be reframed as sequence modeling tasks, where the input is represented by a sequence (in source code and/or natural language), the output is either a sequence or a numeric/categorical value and the goal is to predict the next entity in a sequence based on the previous entities. So, under this perspective, code-related tasks can be addressed through frameworks that were originally proposed for sequence modeling in Natural Language Processing. [1]

The main framework for sequence modeling is the encoder-decoder framework. Specifically, the encoder-decoder framework can be articulated into three main components: encoder, context representation, and decoder, where encoder and decoder typically consist of two Deep Learning models.

2.1 RNN Encoder-Decoder

In the simplest implementation of the framework, plain Recurrent Neural Networks have been employed as encoder and decoder components for sequence modeling tasks. A RNN can learn a probability distribution over a sequence by being trained to predict the next symbol in a sequence. In this case, the output at each time step t is the conditional distribution $p(x_t|x_{t-1},\dots,x_1)$. By combining these probabilities, the probability of the sequence x can be computed: $p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t|x_{t-1},\dots,x_1)$

The RNN Encoder-Decoder architecture [2] learns to encode a variable-length sequence into a fixed-length vector representation and to decode a given fixed-length vector representation back into a variable-length sequence, where input and output sequence lengths may differ. The encoder is an RNN that reads each symbol (or token) of an input sequence x sequentially and updates the hidden state $h_{\langle t\rangle}$ of the RNN accordingly. After reading the end of the sequence, the hidden state of the RNN is a summary $\mathbf{c}$ of the whole input sequence. The decoder is another RNN trained to generate the output sequence by predicting the next symbol y_t. Both $h_{\langle t\rangle}$ and y_t are conditioned on y_{t-1} and on the summary c of the input sequence. The hidden state of the decoder at time t is $\mathbf{h}_{\langle t\rangle} = f(\mathbf{h}_{\langle t-1\rangle}, y_{t-1}, \mathbf{c})$, and the conditional distribution of the next symbol is $P(y_t|y_{t-1}, y_{t-2}, ..., y_1, \mathbf{c}) = g(\mathbf{h}_{\langle t\rangle}, y_{t-1}, \mathbf{c})$. The two components of the proposed RNN Encoder-Decoder are jointly trained to maximize the conditional probability of a correct translation given a source sentence.

However, vanilla RNNs present an important issue: they struggle to capture long-term dependencies. During training, RNNs use Backpropagation Through Time to adjust their parameters (weights and biases) and minimize the prediction error. This is done by computing gradients, taking the derivatives of the loss function with respect to the model's parameters at each time step. These gradients indicate how much each parameter should be adjusted to reduce the error. However, the problem is that as the procedure is propagated backward through time during training, the gradients get increasingly smaller, approaching zero, which means the model does not provide meaningful parameter updates for earlier time steps and thus earlier hidden states have no meaningful effect on later hidden states, meaning long-term dependencies are not effectively learned. This is also known as the vanishing gradient problem [3]. Moreover, since parameter updates are of such a small entity, the vanishing gradient problem extends training time, making RNN models more computationally expensive and hard to train. Finally, since they process words sequentially, RNNs are hard to parallelize.

In order to mitigate the vanishing gradient problem, several techniques have been proposed. Among these:

- Careful initialization of the weights of the RNN to ensure that gradients are not too small initially.
- Limiting the size of gradients during training (Gradient Clipping). If a gradient exceeds a certain threshold, scale it down to prevent it from becoming too small or exploding.
- Use of activation functions that are less prone to vanishing gradients, like the ReLU (Rectified Linear Unit) or variants.
- Implementation of skip connections or residual connections between layers of the RNN to help gradients flow more easily through the network.
- Use of gating mechanisms like Long-Short-Term-Memory (LSTM) and Gated Recurrent Units (GRUs) to control the flow of information. This involves replacing standard RNN cells with LSTM (or GRU) cells, which are designed to better capture long-range dependencies.

- Introduction of attention mechanisms into the RNN architecture to allow the network to focus on relevant parts of the input sequence, reducing the reliance on long-term dependencies.

2.2 LSTM Encoder-Decoder

Focusing on Gated RNNs, they model the keeping and forgetting mechanisms explicitly with sigmoid activation functions, namely gates. An LSTM cell has three gates to control input, output and forgetting, respectively, plus a memory cell state to generate the hidden states. Through the gates, the LSTM cell regulates what information to retain at the next time step and what to discard, then updates the cell state which stores information over time and passes it to the next step, continuing the sequential process. LSTM RNNs are more complex and computationally expensive than vanilla RNNs and, even if they mitigate the issue of vanishing gradients, they are still not entirely free from this problem and present other limitations.

2.3 Attention

Attention mechanisms are more recent advanced techniques introduced to enhance the performance of the encoder-decoder framework and increase its ability to deal with long-range dependencies. The starting point that led to the introduction of such techniques is that the original encoder-decoder approach requires all the necessary information of a source sentence to be compressed into a single fixed-length vector. This

- makes it difficult for the encoder to deal with long sentences, especially if longer than those in the training corpus,
- does not account for the fact that different parts of the source may be more relevant than others at different generation steps, so the decoder would need to focus on those parts, but this becomes hard when the decoder can only access a single context vector and see one representation of the source.

2.3.1 Bahdanau Attention model

Bahdanau et al [4] introduced the attention mechanism. The main characteristic of their novel approach is that it does not force a whole input sentence to be encoded into a single fixed-length vector regardless of its length. The input sentence is instead encoded into a sequence of vectors and the model chooses a subset of these vectors adaptively while decoding the translation. When a word is generated in a translation, the model (soft-)searches for a set of positions in a source sentence where the most relevant information is concentrated and then predicts a target word based on the context vectors associated with these source positions and all the previously generated target words. This approach, where the model learns to align and translate jointly, results in improved performance of sequence modeling tasks with long sentences, where attention is focused on the most relevant source tokens at each step.

The encoder of the Bahdanau model is a bidirectional RNN, which consists of a forward RNN and a backward RNN. The two networks read the input sequence as ordered and in reverse order respectively, and compute a sequence of forward and backward hidden states respectively. For each token, the two overall hidden states (forward and backward) are concatenated in what is called an annotation. Each annotation hi contains information about the whole input sequence, with a strong focus on the parts surrounding the i-th word of the input sequence. The decoder consists of an attention block and an RNN. The attention block computes the context vector c_i for each target word y_i as a weighted sum of these annotations hi:

$$c_i = \sum_{i=1}^{N} \alpha_{ij} h_i,$$

Each weight α_{ij} is obtained through an alignment model (a feedforward neural network), jointly trained with the other parts of the model. The alignment model evaluates the similarity between the input elements near position j and the output at position i by computing a score based on the RNN hidden state s_{i-1} and the annotation h_j . The weight can be seen as a probability that the target word yi is aligned to (or translated from) a source word xj, and so it indicates how relevant its corresponding annotation hj is with respect to the previous hidden state s_{i-1} when determining the next state s_i and producing y_i. After that, the RNN outputs the most probable symbol y_t at the current step.

This introduces an attention mechanism within the decoder, enabling it to selectively focus on parts of the source sentence and alleviating it from the task of encoding all information in the source sentence into a fixed-length vector: information can now be distributed across the sequence of annotations, allowing the decoder to retrieve and utilize it as needed.

One drawback of this kind of approach leveraging attention mechanisms in conjunction with RNNs, is that the recurrent models still inherently imply sequential computation, and this becomes critical at longer sequence lengths and makes it more difficult to learn dependencies between distant positions.

2.3.2 Transformer and Self-Attention

The Transformer architecture, introduced in 2017 with the paper named *Attention is All You Need* [5] allowed for parallel computation and more effective capture of long-range dependencies. It represented a novel sequence-to-sequence modeling approach since it relies uniquely on attention mechanisms to establish global relationships between input and output, without the need for recurrent connections or convolutional layers (unlike previous models). Notably, it leverages a refined form of attention known as self-attention, which supersedes traditional sequence-aligned RNNs in generating representations of input and output sequences.

The transformer model consists of separate encoder and decoder components, both composed of multiple identical layers stacked together. Each layer has two sub-layers: a multi-head self-attention module and a position-wise fully connected feed-forward network. Residual connections are also employed around each sub-layer, followed by layer normalization. The decoder also presents an additional third sub-layer inserted between the multi-head self-attention and the position-wise FFN. This third sub-layer, namely cross-attention module, performs multi-head attention over the output of the encoder stack, allowing for the incorporation of information from the encoder. Moreover, the self-attention sub-layer in the decoder is modified in order to prevent positions from attending to subsequent ones (this procedure is called *masking*). This ensures that predictions for a position are only based on known outputs at preceding positions, and maintains the auto-regressive property of the model. The following sections provide a more explanatory

insight into how the Transformer and its self-attention mechanism work:

Scaled Dot-Product Attention and Multi-Head Attention

[Scaled Dot-Product Attention and Multi-Head Attention] Multi-head self-attention refers to the self-attention mechanism being performed multiple times in parallel in the Transformer architecture; each self-attention module is called a head. The specific type of attention used for the Transformer model is Scaled Dot-Product Attention. The following is an insight on how it works.

During training, the model learns a new representation of the input as a set of Query, Key, and Value vectors, that serve specific purposes within the attention mechanism.

- Each Query vector represents the target word for which the context is being determined and it is compared against other words in the sentence.
- Key vectors represent "the other words" to which the query is compared. The dot product of each query vector with all the key vectors is computed to get an attention score that will reflect the relevance of the word associated with each key to the word of interest represented by the query. The dot products are then scaled by $\sqrt{d_k}$ (where d_k is the dimension of the queries and keys) and are applied a softmax function.
- After calculating the similarity scores using query and key vectors, the system computes a weighted sum of the value vectors to create the final contextual representation of the target word. Value vectors hold the contextual information of each word and the weight assigned to each value is the attention score of the query with the corresponding key.

Queries, keys and values are organized into matrices called Q, K, and V, respectively. The matrix of the outputs for Scaled Dot Product Attention is

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

The use of these three matrices together enables the self-attention mechanism to adeptly capture the interconnections and dependencies among words within a sentence.

Each attention head has its own learned linearly projected versions of Q, K, and V that it applies to the input embeddings to get its queries, keys, and values. Because these transformations are learned separately, each head can potentially learn to focus on different types of relationships and patterns in the input data simultaneously. The output values of the heads are then concatenated to form a unified representation, and projected again, resulting in final output values. Multi-head attention allows the model to jointly deal with information from multiple representation subspaces at different positions.

Position-wise Fully Connected Feed-Forward Networks

[Position-wise Fully Connected Feed-Forward Networks] Each layer in the encoder and decoder has a fully connected feed-forward neural network that is applied independently to each position within the sequence to encode context at each position into higher-level representations [6]. It consists of two linear transformations with a ReLU activation function in between.

Residual Connections and Layer Normalization

[Residual Connections and Layer Normalization] Each of the sub-layers in the encoder and decoder layers has a residual connection around itself, followed by layer normalization. The output of each sub-layer is computed as $LayerNorm(x + Sublayer(x))$, where x represents the input to the sub-layer and $Sublayer(x)$ is the function implemented by the sub-layer itself. This helps in mitigating the vanishing gradient problem and in faster and more stable training.

Input Embeddings and Positional Encoding

[Input Embeddings and Positional Encoding] Each word in the input sentence is converted into word embeddings. Moreover, since the model does not incorporate recurrence nor convolution, it does not inherently include a notion of position within a sequence. So, in order for the model to be able to take into account the order of words in a sequence and attend to relative distances between tokens when computing the attention scores, a positional encoding vector is created for each position in the sequence through modified versions of sine and cosine functions. These positional encoding vectors are then added to the input embeddings before they are passed to the encoder and decoder stacks.

So in essence, the attention mechanism originally introduced by the paper *Neural Machine Translation by Jointly Learning to Align and Translate* [4] was primarily focused on facilitating the alignment of words across input and output sequences in sequence-to-sequence tasks such as machine translation. This mechanism allowed the model to attend to relevant parts of the input sequence when generating each token in the output sequence.

On the other hand, self-attention offers a more nuanced approach: it allows to capture relationships between words within the same sequence, considering the entire input sequence simultaneously. By doing so, it fosters a richer comprehension of the underlying meaning, patterns and contextual information of the input.

The advantages brought about by the Transformer architecture are

- Parallel Processing: Transformers process all inputs in parallel, making them significantly faster
- Capturing Long-Range Dependencies and relevance of time steps for a particular time step.
- Interpretability: Transformers offer interpretability through attention weights, enabling insights into which parts (timesteps) of the input sequence are considered more important for each prediction.

2.4 Large Language Models

Since the Transformer architecture revolutionized the field of sequence modeling, pre-trained language models (PLMs) have been proposed by pre-training Transformer models over large-scale corpora. These models emerged as a versatile and powerful tool for a wide range of NLP tasks, showing a great capability to learn rich representations of language that encapsulate semantic relationships, syntactic structures, and contextual nuances. The research community also observed that scaling up PLMs' parameter size leads to further improved capacity and surprising emergent abilities in solving complex tasks. Then the term "large language models (LLMs)" was coined to identify these large-sized PLMs.

Currently, Large Language Models (LLMs) are mainly built upon the Transformer architecture, since it allows to scale to hundreds or thousands of billions of parameters, due to its excellent parallelizability and capacity. [7]

LLMs can be categorized into three major architectural types

2.4.1 Encoder-decoder LLMs

Encoder-decoder LLMs incorporate both encoder and decoder as separate modules.

- The encoder processes the input sequence of tokens and encodes it into a hidden representation that captures the underlying structure and semantics and acts as an intermediary language, bridging the gap between input and output formats.
- The decoder receives this hidden state and performs cross-attention to generate the output sequence autoregressively, translating the abstract representation into concrete and contextually relevant expressions.

This architecture offers flexible training strategies and is effective for tasks requiring sequence-to-sequence mapping, such as translation, text summarization, dialogue generation, and question-answering. Within the field of SE, this ability has been successfully applied to tasks like code summarization. The encoder's ability to comprehend and represent both the structure and semantics of code is crucial and allows the decoder to translate this understanding into concise, human-readable summaries.

Examples of Encoder-decoder models adapted for code understanding and generation are PLBART, CodeT5, its advancement CodeT5+, and AlphaCode, which show the architecture's versatility in addressing diverse software engineering tasks [8]. However, there are currently only a few large language models based on the encoder-decoder architecture, such as Flan-T5 and CodeT5+. [9]

2.4.2 Encoder-only LLMs

[8] [7] This type of architecture utilizes only the encoder component, whose function is to process and encode the input sentence into a hidden representation, capturing the relationships between words and the overall context of the sentence. Notable implementations of encoder-only LLMs include BERT (Bidirectional Encoder Representations from Transformers) and its variants. As the name suggests, BERT uses a bidirectional attention mechanism that simultaneously considers the left and right context of each word during training. In the field of Software Engineering, CodeBERT, GraphCodeBERT, RoBERTa, and ALBERT have been widely employed, and specialized models such as BERTOverflow and CodeRetriever have been specifically developed for SE applications. These models leverage the program structure, introducing new pre-training tasks, or engaging new modalities, thereby improving the architecture's application to code-related tasks. These models have shown efficacy in tasks requiring a nuanced understanding of the entire sentence or code snippet, including code review, bug report understanding, and named entity recognition of code entities.

2.4.3 Decoder-only LLMs

[8] [7] [9] Decoder-only LLMs exclusively leverage the decoder module to generate the output, following a distinct training paradigm that emphasizes sequential prediction. Unlike the encoder-decoder architecture, where the encoder processes the input, the decoder-only architecture begins with an initial state and predicts subsequent tokens, gradually building the output sequence. This approach relies heavily on the model's ability to understand and anticipate language structure, syntax, and context. The models implementing this architecture can generally perform downstream tasks from a few examples or simple instructions without adding prediction heads or fine-tuning, making them valuable tools in SE research. Decoder-only models are divided into Causal and Prefix Decoder architectures.

- **The Causal Decoder architecture** is commonly implemented as a stack of decoder layers, where input and output tokens are processed in the same fashion through the decoder. It incorporates a unidirectional attention mask (diagonal mask matrix), to guarantee that each input token can only attend to and access information from preceding tokens and itself. This ensures a unidirectional and autoregressive generation process. The GPT-series models represent the major implementations of this architecture. Specifically, it was GPT-3, with the increase in model size and the amount of data used for pre-training compared to previous models, to showcase the power of this architecture, including remarkable in-context learning and few-shot capabilities. Among the LLMs leveraging this type of architecture, are PaLM, LLaMA, Bard, OPT, and Bloom. Moreover, also specialized versions like CodeGPT, InstructGPT, Codex, Copilot, and others have been fine-tuned for specific tasks in SE.

- **The Prefix Decoder architecture** consists of decoder layers too, and the main difference with respect to the previous architecture lies in the attention mechanism. The prefix decoder revises the masking mechanism of causal decoders in order to enable bidirectional attention over the prefix tokens, whereas unidirectional attention is applied to the generated tokens. Similarly to the encoder-decoder architecture, this combination of attention mechanisms allows the prefix decoders to bidirectionally encode the prefix sequence, thus incorporating information from both preceding and succeeding tokens, and to autoregressively predict the output tokens one by one, ensuring a unidirectional flow of information during the generation process. Representative LLMs based on the prefix decoder architecture include GLM-130B and U-PaLM.

2.4.4 Mixture of Experts

[7] Encoder-decoder, prefix decoder and causal decoder architectures can be further extended via the mixture-of-experts (MoE) scaling, in which a subset of neural network weights for each input are sparsely activated (examples are Switch Transformer and GLaM). MoE is a flexible way to scale up the model parameter while maintaining a constant computational cost. It has been shown that substantial performance improvement can be observed by increasing either the number of experts or the total parameter size. However, training large MoE models may suffer from instability issues due to the complex, hard-switching nature of the routing operation. To enhance the training stability of MoE-based language models, techniques such as selectively using high-precision tensors in the routing module or initializing the model with a smaller range have been introduced.

As stated by Hou et al. [8], years from 2020 to 2023 saw notable transitions in the preferred architecture of LLMs and their application within Software Engineering tasks. Specifically, year 2023 marked the affirmation of the decoder-only architecture as the dominant choice for LLMs in the field of SE. This rapid shift can be attributed, along with the need for minimal fine-tuning, to the generative prowess of decoder-only LLMs in producing code that is not only syntactically accurate but also functionally meaningful. Their adeptness in understanding code context made them a favored option.

2.5 Usage scenarios of LLMs for SE Tasks

The key applications of LLM4SE encompass a diverse range of SE tasks, that can be categorized into six main areas, following the phases of the Software Development Life Cycle. These areas are: requirements engineering, software design, software development, software quality assurance, software maintenance, and software management.

Table 2.1 reports specific SE tasks addressed with LLMs, within the different areas.

SE Area	Tasks	
Requirements engineering	Anaphoric ambiguity treatment	Requirements term identification
	Requirements classification	Requirement detection
	Requirement analysis and evaluation	Traceability automation
Software design	GUI retrieval	Software safety analysis
	Rapid prototyping	
Software development	Code generation	Code completion
	Code summarization	Code understanding
	Code search	Program synthesis
	Code representation	Code comment generation
	Method name generation	API recommendation
	API synthesis	API entity and relation extraction
	...	
Software quality assurance	Test generation	Vulnerability detection
	Test automation	Verification
	Bug localization	...
Software maintenance	Program repair	Code review
	Debugging	Bug report analysis
	Code clone detection	Logging
	Sentiment analysis	Vulnerability repair
	Traceability recovery	...
Software management	Effort estimation	

Table 2.1: Software Engineering tasks addressed with LLMs

The two primary domains of application are software development and maintenance, with code generation and program repair as the most explored tasks. However, LLMs also hold great promise in aiding requirements engineering (RE), an area that is so far still under-represented in research. Therefore, the following chapter is dedicated to exploring the application of LLMs to the field of RE.

Chapter 3

LLMs for Requirements Engineering

3.1 Exploratory Literature Review Approach

This chapter presents an exploratory literature review that was conducted focusing on the application of LLMs to Requirements Engineering tasks.

3.1.1 Research Questions

This review aims to answer the following research questions:

- RQ1: How can LLMs be leveraged to streamline the requirements elicitation process?
- RQ2: How can LLMs be leveraged for the conversion of requirements into structured formats and their categorization into functional and non-functional?
- RQ3: How can LLMs be leveraged for enhancing requirements completeness and managing ambiguity?
- RQ4: How can LLMs be leveraged for enhancing requirements traceability?
- RQ5: What is the potential role of LLMs in automating API recommendation?

3.1.2 Search Strategy

The coverage of literature is non-exhaustive and is a set of research articles that are retrieved by search queries and filtered by inclusion and exclusion criteria.

Keywords and Search String

Two sets of keywords were identified, one being RE-related, while the other LLM-related. The final search string is as follows: *(“Requirements Engineering” OR "Requirements elicitation" OR "requirements traceability" OR "Software traceability" OR “Trace links” OR "Traceability links" OR "traceability problem" OR "requirements classification" OR “Formal Requirements” OR “Requirement Forecasting” OR "requirements completeness" OR "requirements ambiguity" OR "requirements verification" OR "natural-language requirements" OR "API recommend*") AND*

(LLM OR "Language Model*" OR "Pre-train*" OR "Natural Language Processing" OR NLP OR AI OR Transformer OR BERT OR Codex OR GPT* OR "In-Context Learning" OR "Transfer Learning" OR "Code completion" OR "Code generation" OR prompt* OR “prompt pattern” OR “prompt engineering” OR “Chain-of-Thought” OR “zero-shot prompting” OR “few-shot prompting”)*

Due to the differences across the search functions of the selected search engines, there may be some differences among the actual search strings. The search was limited to years from 2021 to 2024.

Search Datasets

Four databases, including Scopus, IEEE Xplore, Springer Link, and ACM Digital Library were adopted for the search. A total of 2683 results were collected, organized as follows:

- Scopus: 550 results. The following filters were used for the search: TITLE-ABS-KEY, English language, Computer Science and Engineering as Subject area; Conference paper, Article, Conference review, Review as Document type
- IEEE Xplore: 779 results. textitAll metadata was selected for the search
- Springer Link: 724 results. The following filters were used for the search: English language, Computer Science and Engineering as discipline; article as content type.
- ACM Digital Library: 630 results. The textitresearch article filter was selected.

Exclusion Criteria	Inclusion Criteria
Short papers whose number of pages is less than or equal to 7	The paper has accessible full-text
Non-English written literature	The paper is written in English
Duplicate papers or similar studies with different versions from the same authors	The paper claims that the study involves an RE task
Literature whose primary focus was not to study the application of LLM- or deep learning-based techniques to requirements engineering tasks. For example, literature focusing on electric vehicles, diabetes or other diseases or disorders, environment, smart cities,...	The paper claims that an LLM is used

Table 3.1: Inclusion and Exclusion criteria for study selection tasks addressed with LLMs

3.2 Results

3.2.1 RQ1: How can LLMs be leveraged to streamline the requirements elicitation process?

Requirements elicitation is a critical step in the software development life cycle as it ensures that the final system meets users’ and other stakeholders’ needs and expectations. Gathering requirements is a process that has traditionally heavily relied on manual techniques like interviews, questionnaires, and meetings, resulting in an extensively time-consuming effort, also prone to human error. The advent of LLMs offers a promising path to automate and streamline

RQ	Keywords	selected studies
How can LLMs be leveraged to streamline the requirements elicitation process?	(“Requirements elicitation” OR "requirements generation" OR "generating requirements") AND ("LLM" OR "Language Model*" OR "Pre-train*" OR "Natural Language Processing" OR "NLP" OR "AI" OR "Transformer" OR "BERT" OR "Codex" OR "*GPT*" OR "In-Context Learning" OR "Transfer Learning" OR "Code completion" OR "Code generation")	[10, 11, 12, 13, 14, 15]
How can LLMs be leveraged for the conversion of requirements into structured formats and their categorization into functional and non-functional?	(“Requirements classification” OR “Formal Requirements” OR “Requirement Forecasting”) AND ("LLM" OR "Language Model*" OR "Pre-train*" OR "Natural Language Processing" OR "NLP" OR "AI" OR "Transformer" OR "BERT" OR "Codex" OR "*GPT*" OR "In-Context Learning" OR "Transfer Learning" OR "Code completion" OR "Code generation")	[16, 17, 18, 19, 20, 21, 22, 23, 24]
How can LLMs be leveraged for enhancing requirements completeness and managing ambiguity?	(“Requirements completeness” OR "requirements verification" OR “Requirements ambiguity” OR "natural-language requirements") AND (LLM OR LLMs OR “Language Model” OR “Language Models” OR "Pre-training" OR "Natural Language Processing" OR NLP OR AI OR Transformer OR BERT OR Codex OR gpt OR ChatGPT OR "In-Context Learning" OR "Transfer Learning" OR "Code completion" OR "Code generation")	[25, 26, 27, 28, 29, 30, 31]
How can LLMs be leveraged for enhancing requirements traceability?	("Requirements traceability" OR "Software traceability" OR “Trace links” OR "Traceability links" OR "traceability problem") AND (LLM OR LLMs OR “Language Model” OR “Language Models” OR "Pre-training" OR "Natural Language Processing" OR NLP OR AI OR Transformer OR BERT OR Codex OR gpt OR ChatGPT OR "In-Context Learning" OR "Transfer Learning" OR "Code completion" OR "Code generation")	[32, 33, 34, 35, 36, 37, 38, 39, 40, 41]
What is the potential role of LLMs for automating API recommendation?	("API Recommendation Techniques" OR "API recommendation" OR "API recommendations" OR "API recommender" OR "API recommendation system" OR "API recommended practice" OR "API recommenders") AND (LLM OR LLMs OR “Language Model” OR “Language Models” OR "Pre-training" OR "Natural Language Processing" OR NLP OR AI OR Transformer OR BERT OR Codex OR gpt OR ChatGPT OR "In-Context Learning" OR "Transfer Learning" OR "Code completion" OR "Code generation")	[**huang2023let** , 42, 43, 45, 46, 47, 48, 49, 50, 51]

Table 3.2: reports Research Questions along with the specific keywords and selected studies

this process, enhancing efficiency and accuracy. RQ1 explores how LLMs can be leveraged to streamline the requirements elicitation process by synthesizing findings from several relevant studies. Applications include requirement generation, interview script creation, and requirements extraction from diverse sources.

Ronanki et al. [10] leverage ChatGPT to assist in the generation of high-quality initial requirements based on a requirements elicitation questionnaire. The example use case is the realization of a Trustworthy AI system, and the questionnaire addresses aspects including accuracy, robustness, fairness, transparency, privacy, and human oversight. The same questions are posed to both RE experts and ChatGPT, and responses are evaluated by other experts without knowledge about their source. The quality of requirements is assessed based on seven attributes.

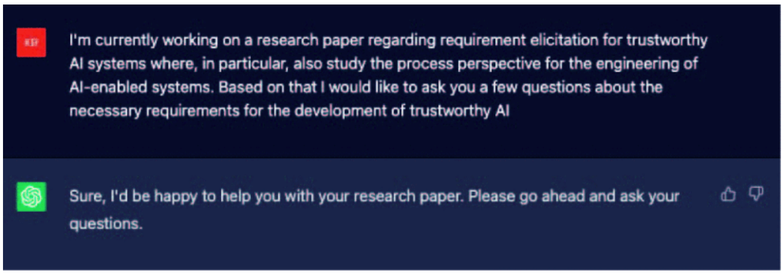


Figure 3.1: Context Prompt

The results reveal that ChatGPT-generated requirements are generally comparable to those created by human experts in terms of quality, scoring high on attributes such as Abstraction, Atomicity, Consistency, Correctness, and especially Understandability. However, lower performance is observed in Unambiguity and Feasibility compared to human-crafted requirements, and ChatGPT also struggles with generating requirements belonging to different areas than those directly addressed by the questionnaire (such as accountability and regulations requirements in the context of trustworthy AI).

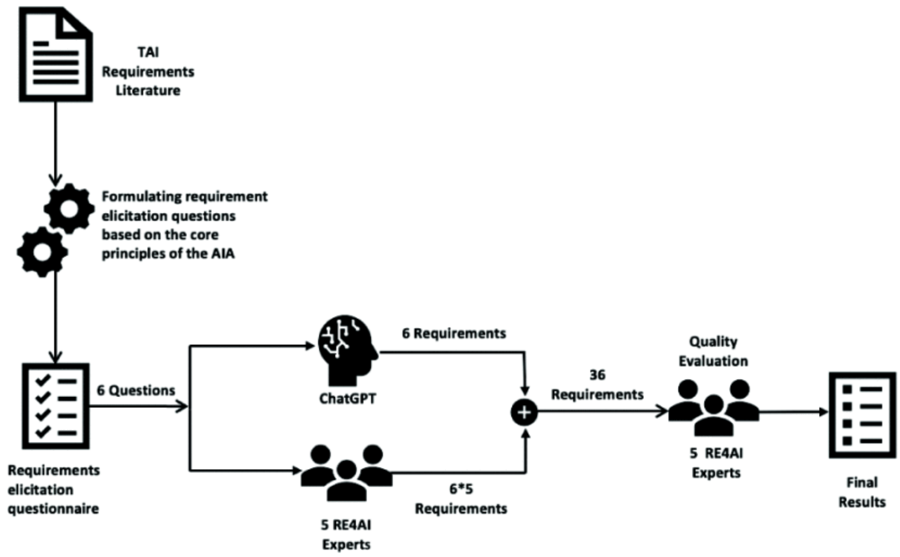


Figure 3.2: Overview of the methodology used.

The authors also acknowledge the issue of "hallucinations" in LLM outputs, where the model may generate plausible but incorrect information that cannot be validated from the source. Overall, ChatGPT-generated requirements are not yet entirely suitable on their own for use in real projects, but despite these limitations, the results suggest that LLMs like ChatGPT can serve as a supportive tool, providing a baseline of requirements for further refinement through human expertise, allowing for time and effort saving in the initial stages of requirements elicitation.

Another area where LLMs can intervene in the requirements elicitation process is the automation of requirements elicitation interview scripts generation.

To approach the task, *Görer et al.*[11] propose a method to derive the graph representation of an interactive interview script by parsing its HTML file. Indeed, in interactive interview scripts, multiple paths are possible, based on the analyst's questions. The nodes of the graph are the stakeholder's responses, while the edges are the analyst's questions, and each node is connected to three edges representing the available choices for the analyst's question following a stakeholder response. The graph representation provides a structured visualization of the interview process.

The study combines two strategies for the generation of interview scripts: batch and iterative generation.

In the batch generation approach, the LLM is prompted to create the entire interview script in one go. Two types of prompting techniques are tested for this approach: few-shot and chain-of-thought prompting. For the few-shot prompting setting, the initial prompt defines the task and business domain and also instructs the LLM to incorporate errors from a list of common mistakes in the questions to account for the mistakes that analysts can make during the interview process. Next, multiple linear paths derived from the interactive graphs are provided to the LLM through subsequent prompts as few-shot examples. The reason for this is that the entire graph conversation is hard to process and increases task's complexity making it challenging to maintain coherence and logical flow in the output. Moreover, to achieve comprehensive coverage of the analyst mistakes, three linear interview scripts are generated: one without errors and the other

Stakeholder1:	I'm doing well thank you for asking!
Analyst2a:	In this meeting, I'd like to ask you a few questions about your company.
Stakeholder2a:	Okay, go ahead.
Analyst2b:	I want to ask you some questions about this project that you want us to work on, can you tell me a little bit about yourself?
Stakeholder2b:	So I'm the owner of the cool ski resorts, we have three resorts in the state of Colorado, and I want to improve the operations of my company.
Analyst2c:	I'm the requirements analyst of our company and I'd like to ask you some questions about your business so that we can build a product that suits for business needs. First, can you tell me about you and your functions in the company?
Stakeholder2c:	I'm the owner of the cool ski resorts, I have three locations in Colorado, I overlook all operations that are performed in my company. I want to improve my business to get a competitive edge in the market.

Figure 3.3: An interactive interview script

two with specific types of mistakes, making sure to include at least one example of each error type. On the other hand, the chain-of-thought technique breaks the interview script generation task down into smaller steps, where each prompt takes the information provided by the previous one and the output is multi-step. In this setting, the LLM is first prompted to propose a business case study based on a given business and define the potential features of the target product. The next prompts ask the LLM to select a specified number of the provided features and then generate the interview script based on the selected features, specifying the minimum number of turns and reminding to incorporate mistakes.

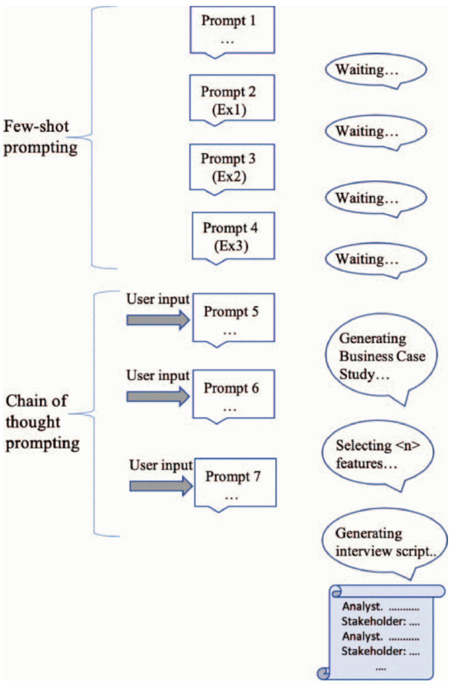


Figure 3.4: The prompting structure for batch interview script generation.

In the iterative generation approach, the LLM builds the script incrementally turn by turn, based on the previously generated ones, which enables greater control and refinement. For example, it allows to specify the desired mistake type to include in the subsequent analyst

ID	Mistake
M1	No rapport with stakeholder
M2	Asking long question
M3	Lack of preparation
M4	Asking technical question
M5	Influencing stakeholder
M6	Asking vague question
M7	Asking stakeholder for solution
M8	Not asking for existing system
M9	Unnatural dialogue style
M10	Ignoring other stakeholders
M11	Asking unnecessary question
M12	Incorrect ending of interview
M13	Did not provide short summary

Figure 3.5: The mistakes induced in the interview script.

question, to request explanations and regeneration of the response. The initial steps are the same as for the batch generation approach, but in the last prompt, the LLM is asked to generate the next interview turn based on the previous turns, i.e. the partial interview script.

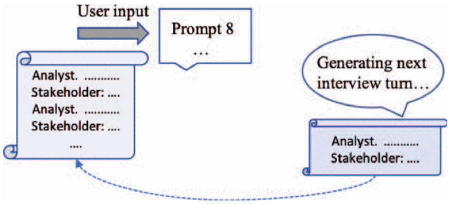


Figure 3.6: Prompting structure for iterative interview script generation.

The generated scripts are evaluated based on their completeness in terms of coverage of given features of business use cases, coherence across conversation turns, and accuracy in addressing analyst mistakes. The comparison involving Bard and ChatGPT shows that the latter outperforms the former in generating more complete and coherent scripts for the given product features in the case of batch generation, while Bard performs better in iterative generation, with more accurate interview turns for specific mistake types and more formal explanations of how the mistakes have been incorporated into the analyst's questions.

The models face limitations in interpreting and implementing some kind of information (specifically, certain mistake types) and show the tendency to forget inputs provided at the beginning of the process, which makes reminding instructions across prompts important. To address these issues and enhance the overall quality of the results, researchers recommend working with smaller tasks by splitting the original ones into steps, refining prompts to provide more precise instructions to the LLMs, and performing fine-tuning on domain-specific datasets to improve performance.

Despite these challenges, the findings suggest that LLMs can assist in automating the generation of requirements elicitation interview scripts, saving time and effort for educators and students and acting as valuable tools for teaching best practices and pitfalls in requirements elicitation. A fundamental observation that needs to be pinpointed is that careful prompt design is crucial for ensuring accuracy and completeness of the output.

Large Language Models have also been studied in their application to infer system requirements from user commands in the context of autonomous driving systems by *Yang et al.*[12]. The integration of such AI models within autonomous systems reduces the dependency on pre-defined rule-based systems and boosts their capability to grasp and interpret both scene dynamics and user intent through a deeper understanding of NL commands of in-cabin users. This human-centric approach enhances the system's alignment with user preferences and behaviors, fostering improved trust, transparency, and decision-making in real-time interactions. The models used in the study include GPT-3.5, GPT-4, CodeLlama-34b-Instruct (a variant of the original Llama model additionally trained on code generation and instruction problems with 34 billion parameters), and Llama-2-70b-Chat (trained mainly on conversational interactions enhanced with human reinforcement learning, and with 70 billion parameters).

The task is a multivariate binary classification task of AD systems requirements where the model has to provide a 'yes' or 'no' answer to whether a command requires any of eight specified modules to achieve autonomy and a step-by-step explanation of the results (the modules include perception, in-cabin monitoring, localization, control, network access, entertainment, human privacy, and traffic laws). Since LLMs are originally trained on vast amounts of general-domain knowledge, the success in adapting them to new tasks also depends heavily on prompting techniques and quality of prompts. Thus, to improve the model's reasoning capabilities and ensure a step-by-step reasoning process with a clear sequence to follow, the prompts are provided with in-context few-shot examples resembling a chain-of-thought and designed with a detailed explanation for each of the eight questions. Two formats for detailed explanation are tested, a step-by-step approach and a consolidated paragraph format, essentially differing in the use or omission of the word 'step:' before detailing each question, which causes the prompt to be more or less structured.

LLM prompt conditioning and prompt design with user examples	
a) LLM Conditioning	You'll receive a command message for a self-driving vehicle. Follow these steps to respond: Step 1: First decide whether the external perception system is required for this command. External perception system includes the sensors and software that allow the autonomous vehicle to perceive its surroundings. It typically includes cameras, lidar, radar, and other sensors to detect objects, pedestrians, other vehicles, road conditions, and traffic signs/signals. For example, any movement, sense or detect the surroundings. Step 2: Answer "Is in-cabin monitoring required?" in-cabin monitoring involves cameras, thermometers, or other sensors placed inside the vehicle's cabin to monitor the state of occupants and other conditions. It includes everything in-cable system, for example, seats, windows, doors, multimedia system, alert system, etc. ... ... Step 8: Answer "Is there a possibility of violating traffic laws?" Violating traffic laws refers to any action performed by the vehicle that goes against the established traffic regulations of the region. An autonomous vehicle's system is typically designed to adhere strictly to traffic laws. It includes anything risk command. For example, related to the traffic laws, like speed, traffic light, emergency action etc. Answer the 8 questions use the following format: Step 1: 'Yes' or 'No' <step 1 reasoning> Step 2: 'Yes' or 'No' <step 2 reasoning> ... ... Step 8: 'Yes' or 'No' <step 8 reasoning> Response to user: Output is [A1 A2 A3 A4 A5 A6 A7 A8]. Replace A1-A8 with 1 for 'Yes' and 0 for 'No'.
b) User Assistant	[*Provide few-shots for LLM to learn. Show one shot as an example below.*] Call my friend Carol. Explaination: Step 1: No, it doesn't involve any movement, sense or detect the surrounding. Step 2: Yes, it requires to use the in-cabin multimedia to call the people. ... ... Step 8: No, it is not a risky command. Therefore, the output should be : [0 1 0 0 1 1 1 0].
c) User GPT-4 [21]	[*Test on real commands.*] Make an illegal U-turn here. Explanation: Step 1: Yes, it needs to sense or detect the surrounding for making a u-turn. Step 2: No, it doesn't involve anything inside the vehicle's in-cabin system, like seats, windows, doors, multimedia system, alert system, etc. ... ... Step 8: Yes, it is a risky command and it is illegal to make a u-turn Therefore, the output should be : [1 0 1 1 0 0 0 1].

Figure 3.7: View of the approach. Steps are in orange (each step contains one question), user inputs are in purple, explanations are in blue, and the corresponding LLM classification output is in green. a) LLM initial conditioning of several steps in the form of a prompt. b) Few-shot examples resembling a chain-of-thought to improve reasoning. c) Example with test commands on GPT-4.

Performance is evaluated on a dataset of in-cabin user commands at accuracy in individual questions (question-level accuracy) and on the entirety of questions for a given command (command-level accuracy). LLMs are compared against two baselines: one employs a random guessing strategy, and the other is a rule-based method that identifies specific keywords in the user command to determine system requirements (these rules are generated by ChatGPT-4, augmented with the Advanced Data Analysis plug-in). LLMs show notably higher accuracy, especially at command-level, with the GPT series outperforming the Llama models. Concerning question-level accuracy, GPT models, especially GPT-4, show some inconsistency and lower accuracy with in-cabin monitoring, misclassifying many commands as requiring this module when in ground truth they don't. The reason, however, is likely due to a vague definition of in-cabin monitoring in the context of the study, therefore GPT's outputs cannot be entirely considered erroneous.

The paper also finds that while each prompting technique enhances performance, the few-shot method outperforms the two detailed explanation methods, and combining detailed explanations with few-shot examples yields the best performance improvement, further increasing the model's ability to understand complex user inputs. Concerning the optimal number of few-shot examples, including two of them appears the best choice, whereas further increasing this number results in a subsequently lower impact on the final results, suggesting the existence of a saturation threshold or limitations. Incorporating real-time human feedback is identified as a future direction to further enhance the LLM's reasoning capabilities and especially their alignment with human-centric behaviours and preferences.

Requirements can also be extracted from texts like engineering documents. Traditional rule-based techniques for information extraction offer great transparency but lack the ability to capture the linguistic variety of texts, while machine learning and deep learning methods achieve higher accuracy but are limited in their generalization ability on unknown texts.

Luttmer et al.[13] compare different techniques for requirements extraction from engineering standards, where the task is a classification of text into either the 'requirement' or 'information'. The comparison includes a rule-based system; supervised learning using SVM, MNB, and CNN; unsupervised learning using German BERT fine-tuned on examples of requirements and information and German BERT in a zero-shot setting where classification is purely based on the pre-trained model and the definition of the labels.

Existing benchmark datasets such as PURE or PROMISE could not be reused for the evaluation since this research focuses on German engineering standards documents in XML format. Thus a dataset combining documents from various application domains and standard types is created and from this, the requirements dataset is generated using an XML parser and NLP techniques. Sections and text blocks that may contain requirements are filtered, followed by dividing texts into sentences as well as word sequences, which are saved as separate dataset entries. The entries are manually classified in requirements and information, and since there is a bias towards information elements, undersampling is applied.

The rule-based system is based on pattern recognition: if a match exists between an input sentence and a defined rule or pattern, the sentence is classified as a requirement, otherwise as information. The rules are derived from analyzing existing requirements in standards documents; some use regular expressions, others are more advanced and require sentence preprocessing using NLP methods like tokenization, lemmatization, and PoS tagging. The supervised learning approaches preprocess the input text before training as well as for each classification task, converting individual words into vectors with techniques such as the bag-of-words or the TF-IDF method. Training for these models is performed once, so the consistency of results is not assessed.

The results of the evaluation show that requirements can be automatically extracted from engineering standards with high quality. SVM and fine-tuned BERT achieve the best F1-score and balance between recall and precision, while the rule-based system and the other supervised learning techniques (CNN and MNB) are close behind in terms of F1-score but tend to favor precision over recall and viceversa respectively. The lowest performance is obtained by BERT in the zero-shot setting. The authors therefore conclude that if transparency is considered critical, rule-based systems should be preferred, but in cases where recall should be prioritized, like the automatic requirements extraction from standards documents, learning-based techniques are more suitable, as it is more convenient to filter out incorrect classifications from a list of possible requirements. Between learning-based methods, unsupervised learning approaches offer significant advantages in aspects such as implementation and adaptation efforts, primarily because of the labor-intensive and time-consuming process of generating labeled training and validation data in supervised techniques.

Another source for the extraction of requirements are app reviews, that present unique challenges like, for example, their unstructured format, grammatical inaccuracies, huge presence of irrelevant information, and multitude of app domains with their own set of features and vocabulary. As usual, common rule-based approaches are not suitable for addressing these challenges mainly due to their poor generalization ability, lack of contextual understanding, and dependency on external sources, while embedding-based approaches based on techniques such as Word2Vec encounter semantic limitations, due to their static representation of word embeddings. *De Araújo et al.*[14] propose RE-BERT to extract requirements from app reviews addressing these issues and increasing efficiency and accuracy. The task is framed as a token classification problem using BIO labeling and leveraging BERT for contextual word embeddings.

In the training process, the model generates training data by splitting reviews into individual tokens, each of which is assigned a BIO label based on whether it's part of a requirement (B indicates the token represents the beginning of a software requirement; I indicates the token is inside a software requirement; O indicates the token is outside a software requirement in the sentence). Given a sequence $\mathbf{x}$ of tokens representing an app review, the goal is to find a sequence $\mathbf{s}$ of BIO labels, one for each token, that maximizes the posterior probability $p(\mathbf{s}|\mathbf{x})$, which can be estimated via sequence labeling models like BERT. In the context of software requirements extraction, given the token sequence of the review, BERT first generates a version of the input where approximately 15% of the words are randomly masked and then predicts the masked tokens so as to maximize the likelihood of these predictions.

A key aspect of the proposed approach is that, even though BERT can learn contextual word embeddings from long-term dependencies within and between sentences, RE-BERT leverages a training strategy that emphasizes local context. This focus on local context is intended to improve the accuracy of software requirements extraction by reducing noise from tokens that are not directly related to the requirements, based on the observation that tokens closer to a requirement are generally more semantically relevant for the requirement's extraction, whereas review tokens distant from requirement tokens can negatively impact the extraction stage. The local context mechanism is implemented as part of the fine-tuning process, therefore it is not used to learn the initial embeddings but is applied during fine-tuning to adjust the weights of tokens in the classification process, influencing how the model assigns BIO labels by focusing on the most relevant tokens. The model is fine-tuned to find significant correlations between the sequence of tokens in a review $\mathbf{x}$ and a subsequence of tokens from $\mathbf{x}$ representing the software requirement adopting a relative position distance (RD) measure to assign decaying weights to tokens with progressively higher RD value and based on a threshold parameter.

The model uses both the local context features representation and the global context features

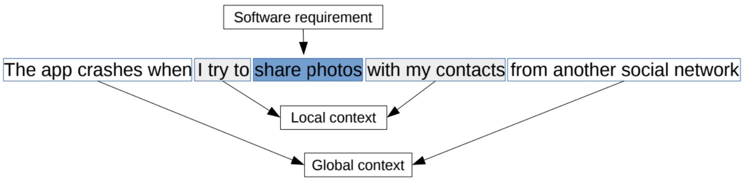


Figure 3.8: Example of an app review organized in (1) software requirement; (2) global context; and (3) local context.

representation, which involves general relationships between the review and the software requirement and is obtained by BERT's existing next-sentence prediction training strategy. These two representations are concatenated into a single vector and used as input for the dense layer of the neural network, followed by the Multi-Head Self-Attention (MHSA) encoder of the Transformer architecture and the final output layer, where a softmax function predicts the BIO label for each token. Then the cross-entropy loss function is used for training the network. After training the model and in order to extract software requirements, RE-BERT first identifies the BIO label of each token in the review text, then extracts all tokens classified as B or I as software requirements and combines consecutive B or I tokens as a single extracted software requirement.

Another essential step in RE-BERT is the use of a cross-domain training strategy, where BERT is trained on pre-existing labeled data from multiple app domains, making it able to extract software requirements from unlabeled review datasets and different domains, without needing specific training for each domain, while also enhancing the generalization capabilities.

For the evaluation of RE-BERT, the authors conduct cross-validation across multiple domains, using reviews from eight different apps. RE-BERT is fine-tuned on labeled data from seven apps and then tested on the eighth app for software requirements extraction, all repeated for each app until all apps are used as test data. RE-BERT is compared to three state-of-the-art methods for software requirements extraction, GuMa, SAFE, and ReUS. These approaches all employ a rule-based information extraction strategy, where a subset of app reviews is analyzed in advance to identify common linguistic patterns associated with text passages describing software requirements, and algorithms are then trained to automatically detect these predefined patterns that represent the requirements.

To determine whether an extracted software requirement is true or false positive, it is compared with software requirements in the annotated dataset as ground truth. To this end, the feature matching method is used, and the models are tested both in an exact matching scenario and a partial matching scenario. In the first case, an error occurs when not all the tokens in the software requirement are identified; in the second case, an error occurs when the model does not identify at least a specified number of tokens.

The experimental results on F1-scores show that RE-BERT outperforms all baselines, especially in the exact matching scenario, and it also performs stably across multiple runs performed for each app, where it remains competitive both in exact and partial matching demonstrating consistent performance and competitive results compared to rule-based models. Moreover, the statistical analysis carried out by the authors outlines no significant difference between the three rule-based methods GuMa, SAFE, and ReUS for the exact matching scenario, while RE-BERT shows a statistically superior performance. For the partial matching scenario, RE-BERT does not display significant differences compared to SAFE but performs statistically better than GuMA and ReUS methods.

These results indicate that the RE-BERT method is a valuable and ready-to-use tool for automating the extraction of software requirements, whereas future work could involve extending RE-BERT to other stages of review mining for requirements engineering, including for example semantic clustering of extracted requirements which will group different variations of the same requirement to standardize them, and polarity classification to enable sentiment analysis based on software requirements.

Another precious source for the requirements elicitation activity is crowd-generated content, like feature requests in issue tracking systems such as GitHub Issue Tracker, along with the related comments submitted by stakeholders. *Wang et al.*[15] propose VOTEBOT for the automatic detection of the comments' stance and summarization of the opinions in feature request discussions, to help in the decision-making process. VOTEBOT is a relation-aware approach that considers different types of relations to achieve the tasks. For stance detection, it extracts the explicit and implicit reply-to relations among comments and incorporates them into the BERT-based classifier. For stance summarization, it incorporates the semantic relation of comment sentences with the feature description and the argumentative relations among comment sentences.

The input of the model is a NL feature request, with description and related comment stream. After preprocessing, the model performs stance detection through reply-to relations retrieval. Explicit reply-to relations are retrieved through pattern matching, while implicit reply-to relations are retrieved using a feedforward model based on textual features, that predicts which comment the current one is most likely to reply to. After this step, the BERT model classifies each comment assigning the stance polarity concerning the feature request. Comment stance generally remains consistent for a specific commenter's role (e.g. Author, Contributor, Collaborator), while in many cases, stance switch is associated with a role switch. Therefore, the commenter's role and most recent parent and child comments are used to represent reply-to relations of the current comment, considering the most recent parent or child comment. The embedding layer of the model is used to represent the commenter's roles with vectors that can be trained jointly with the whole model. The comment text is fed into BERT to obtain its vector representation embedding the semantics and then concatenated with the current comment role, parent role, and child role. The final vector is then input into a dropout layer to avoid overfitting, and into a fully-connected layer which computes the probability vector of polarity labels (i.e., pro/neutral/con) using the cross entropy loss function.

The next stage is deriving the summary sentence for each comment, which is represented by the most informative sentence in the comment. First, semantic relevance is calculated using cosine similarity between sentence vectors and the feature description. Second, argumentative relations measuring the functional role and contribution of every sentence are identified through a neural network with self-attention mechanisms, focusing on patterns like MajorClaim, which typically represents the main opinion in a comment. Finally, graph-based summary extraction is performed: for each comment, a graph is constructed where nodes represent sentences and edges represent the similarity between the two involved sentences. The final centrality combines argumentative relations, semantic similarities between a sentence and other sentences within the comment, and the semantic relevance between a sentence and the feature description. The sentence with highest centrality is selected as the summary of the current comment. Then these summaries of pro and con comments are combined to generate the summaries of the two polarities, which will be used in the debate for a feature request.

VOTEBOT is evaluated on 250 feature requests involving 6598 comments from five popular and widely-used projects hosted on GitHub.

In the stance detection multi-class classification task, VOTEBOT outperforms all other methods used for comparison (CoreNLP, TextCNN, and BERT) by a large margin, achieving an

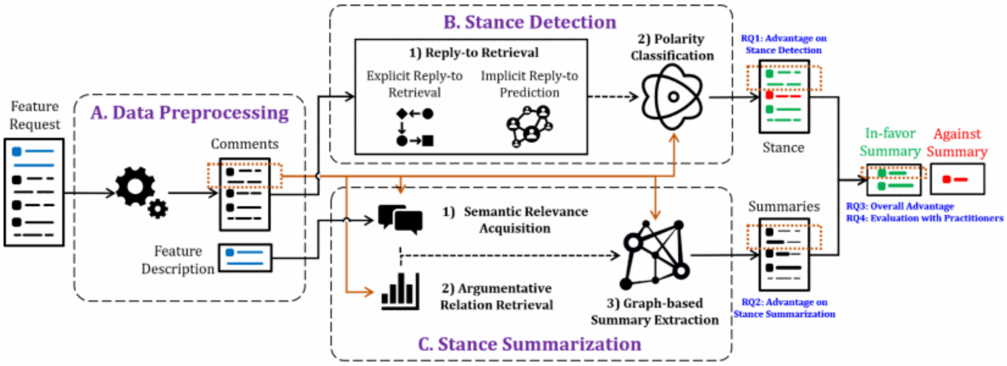


Figure 3.9: Overview VOTEBOT.

overall precision achieved by VOTEBOT is 79.1% while recall is 78.0%. Regarding performance across stance labels, highest precision is achieved on pro labels, while highest recall and F1-score are obtained on neutral labels, indicating that the semantics of neutral comments can be more easily distinguished from pro/con comments by the model, while it is hard to distinguish the semantics between pro and con comments. Moreover, the authors assess the influence of explicit and implicit reply-to relations and the results reveal that they both boost the overall performance regardless of their specific type, especially on pro/con labels, and their contribution overlaps to some extent.

For stance summarization, VOTEBOT achieves 81.20% accuracy, outperforming four commonly used baselines. VOTEBOT also exhibits more consistent and stable performance across the different projects considered for the experiment, as the semantic relevance and argumentative relations features that VOTEBOT incorporates are project-independent. Moreover, the evaluation also compares VOTEBOT variants to assess the influence of different relations on stance summarization. Only considering the structural relationships between comment sentences when determining centrality leads to the worst performance. Introducing semantic relevance increases performance, but the greatest improvement is seen with argumentative relations, which shows that understanding positional and argumentative patterns is more beneficial than just adding semantic relevance. The full VOTEBOT model, which combines both features, achieves the best performance overall.

For the evaluation of the final two stance summaries, the overall performance is influenced by both stance detection and summary sentence selection, with VOTEBOT again significantly outperforming two state-of-the-art baselines. These include OpenIE, which applies sentence matching to detect controversies (used for stance detection) and TextRank for summarizing controversies (used for stance summarization); and ChatEO, which employs textual heuristics to identify opinion-asking questions (used for stance detection) and a deep learning method for extracting answers from conversations (used for stance summarization).

The results confirm the effectiveness and usefulness of VOTEBOT in both experimental setups and real-world practice as demonstrated through an evaluation with practitioners. Future directions may explore the representation and integration of code snippets and images within comment texts, to gain a better understanding of discussions about feature requests.

Requirements elicitation is a crucial step in the software development life cycle as it ensures that the final system meets users' and other stakeholders' needs and expectations. While this activity traditionally relies on time-consuming manual methods, Large language models have paved the way to automate and streamline the process, improving efficiency and accuracy. RQ1 explores the recent findings on how LLMs can assist in requirements elicitation tasks, such as requirements generation, interview script creation, and requirements extraction from diverse sources.

Ronanki et al. [10] investigate the use of ChatGPT to assist in requirements elicitation by generating initial requirements for Trustworthy AI systems. A requirements elicitation questionnaire addressing aspects like accuracy, robustness, fairness, and privacy was posed to both ChatGPT and human experts. The generated requirements were then evaluated by other experts using seven quality attributes. The results show that ChatGPT-generated requirements are generally comparable to human-crafted ones in abstraction, atomicity, consistency, correctness, and understandability, but perform worse in unambiguity and feasibility. While ChatGPT struggles with certain aspects like regulatory requirements and can produce "hallucinations," the study suggests that LLMs can serve as valuable tools for supporting requirements elicitation, especially in the early stages, but human oversight remains essential.

Görer et al.[11] explore how LLMs like ChatGPT and Bard can automate the generation of requirements elicitation interview scripts based on a graph representation of a sample interactive interview script and a list of common analyst mistakes. Two generation approaches are used: batch generation, which creates the entire script at once using few-shot and chain-of-thought prompting, and iterative generation, which builds scripts incrementally turn by turn. The generated scripts are evaluated on completeness, coherence, and accuracy in addressing common mistakes. ChatGPT performs better in batch generation, with more coherent and complete scripts, while Bard scores higher in iterative generation, especially in handling mistakes. Despite limitations in remembering earlier inputs and handling complex interview scenarios, the findings suggest that LLMs can effectively assist in automating the generation of requirements elicitation interview scripts, and further improvements can be made by refining prompts and fine-tuning LLMs on domain-specific data.

Yang et al.[12] explore how LLMs such as GPT-3.5, GPT-4, CodeLlama-34b-Instruct, and Llama-2-70b-Chat, can infer system requirements from user commands in autonomous driving systems. By integrating LLMs, these systems move beyond rule-based approaches to better understand both scene dynamics and user intent through NL commands. The models perform multivariate binary classification, determining if user commands require specific autonomous driving modules like perception, in-cabin monitoring, or traffic law adherence, and test three prompting techniques including few-shot learning and two detailed explanation formats. The results show that LLMs significantly outperform random-guessing and rule-based baselines, with GPT models performing the best overall, though they struggle with in-cabin monitoring, likely due to a vague definition. Combining two few-shot examples with detailed explanations, especially in the step-by-step format, achieves the best performance. The paper suggests that incorporating real-time human feedback could further refine LLM performance and alignment with human behavior in autonomous systems.

Luttmer et al. [13] investigate various techniques for automatic requirements extraction from German engineering standards, focusing on distinguishing between "requirements" and "information". The authors compare a rule-based system, supervised learning using SVM, MNB, CNN, and unsupervised learning using BERT both in the fine-tuned version and a zero-shot setting. A new dataset was created by parsing XML engineering standards, filtering relevant sections, extracting sentences, and manually classifying them. SVM and fine-tuned BERT outperform other methods, achieving F1-scores above 87%, with a balanced precision and recall. In contrast, the rule-based approach also performs well but favors precision over recall. Zero-shot BERT, on the other hand, delivers the lowest performance. The authors suggest rule-based systems for transparency but recommend learning-based methods in scenarios where high recall is preferred, as they allow easier filtering of incorrect classifications. Between learning-based techniques, unsupervised learning offers practical advantages for real-world implementation as it does not require generating vast amounts of labeled training and validation data as supervised methods do.

De Araújo et al.[14] develop RE-BERT to automatically extract software requirements from app reviews, addressing challenges like informal language, grammatical errors, and noise. IRE-BERT leverages BERT's contextual embeddings for improved accuracy and efficiency and frames requirements extraction as a token classification task using BIO labels (B for the beginning of a requirement, I for inside, O for outside). The method emphasizes local context during fine-tuning by assigning higher weights to tokens close to a requirement as they are more semantically relevant for the requirement's extraction. RE-BERT is trained using a cross-domain strategy, leveraging pre-existing labeled data from multiple app domains, which enables it to extract software requirements from unlabeled review datasets and different domains. The experiments reveal that RE-BERT outperforms rule-based methods such as GuMa, SAFE, and ReUS, particularly in exact matching scenarios, showing superior F1-scores across multiple datasets. Its robustness extends to partial matching, making it highly effective for requirements extraction.

Wang et al.[15]propose VOTEBOT to automate stance detection (for/against) and summarization of discussions on feature requests in issue tracking systems like GitHub, enhancing requirements elicitation from this type of crowd-generated content and helping to determine if a feature should be accepted. The method uses a BERT-based classifier, incorporating explicit and implicit reply-to relationships to classify stances as pro, neutral, or con. For summarization, VOTEBOT combines semantic relevance (i.e., how related each comment is to the feature request) with argumentative relations to identify the most relevant sentences and then builds a graph for each comment where edges represent the similarity between two sentences (nodes). The most informative sentence with highest centrality is selected as the summary. This method generates clear summaries of both pro and con perspectives. Upon evaluation, VOTEBOT outperforms all baseline models in precision and recall on stance detection, and 81.2% accuracy in summarization, and demonstrates its effectiveness in simplifying complex discussions and aiding real-world decision-making.

Overall, the analyzed studies show that LLMs can effectively support requirements elicitation tasks including generating requirements, creating interview scripts, and extracting relevant information from various sources. In most cases, the performance of LLM-based approaches is superior or comparable to that of human efforts, rule-based systems, or traditional ML- and DL-based techniques. Although challenges remain, including issues with ambiguity, feasibility, and handling complex scenarios, the incorporation of LLMs holds great promise, especially when combined with human oversight and domain-specific fine-tuning, enabling a powerful synergy between automation and expert input.

3.2.2 RQ2: How can LLMs be leveraged for requirements classification, categorization into functional and non-functional or other categories, and conversion into structured formats?

Requirements classification focuses on the distinction between functional and non-functional requirements or among different classes inside the non-functional category, helping in systematically understanding and managing the scope of a project, identifying key functionalities, and prioritizing features based on their importance and feasibility. Also, requirements are generally expressed in natural language and thus need to be then converted into structured or semi-structured formats (including, for example, user stories or temporal logics). This conversion is essential for several reasons: it translates often ambiguous or high-level stakeholder needs into clear, actionable, and testable specifications; it allows communication and collaboration among teams; it supports traceability throughout the development lifecycle, and ensures that the final product aligns closely with the initial vision and objectives. Over the years, research has explored methods to automate the classification of requirements and conversion into specific formats, initially based on traditional ML and NLP techniques. These approaches suffer from poor generalization on unseen classes and projects; rely on manual pre-processing; neglect structural and syntactic information in requirements text; struggle with incomplete and evolving requirements and require large task-specific labeled datasets for training.

A landmark work in the application of large language models to requirements classification is the NoRBERT model by *Hey et al.* [16], which focuses on addressing the poor generalization of traditional methods and the scarcity of labeled training data. The proposed approach uses BERT to classify requirements on the PROMISE NFR dataset and leverages its transfer learning capability to assess the generalizability on a relabeled version of the PROMISE dataset by *Dalpiaz et al* [52], which distinguishes between functional and quality aspects.

NoRBERT fine-tunes the base and large versions of BERT on a small amount of labeled data for the target task. The input is processed directly by the BERT tokenizer, and the output of the [CLS] token is fed into the classification head (a single layer of linear neurons in a feedforward neural network), to produce final predictions, converted into probabilities using the softmax function. The cross-entropy loss function is used during training to measure the difference between the predicted and true distributions, and the AdamW optimizer updates weights within the network.

NoRBERT was evaluated and compared with state-of-the-art approaches on binary classification of functional and non-functional requirements (Task1), binary and multiclass classification of the four most frequent NFR classes (Task2), multiclass classification of all NFR classes (Task3), and binary classification of functional and quality aspects (Task4).

The evaluation employs various settings, including a 75%-25% train-test split, stratified 10-fold cross-validation, and also project-specific folding strategies (p-fold, project-level cross-validation, and loPo, leave-one-project-out cross-validation) for the evaluation of the generalization on unseen projects, since 10-fold does not account for the existence of different projects with different domains and wording in the dataset. These methods ensure the robustness and generalizability of the model across different datasets and project scenarios, while techniques such as undersampling and oversampling were used to address class imbalance during training, and early stopping was employed to mitigate overfitting and enhance model performance.

For Task 1, NoRBERT is compared with state-of-the-art approaches by *Kurtanović and Maalej* [53] (using automated feature selection and SVM classifier), *Abad et al.*[54] (relying on manual preprocessing and the Naïve Bayes classifier), and *Dekhtyar and Fong* [55] (using word embeddings and a CNN). NoRBERT performs similarly to or outperforms all other models, except for the approach by *Abad et al.*, but has the advantage of not requiring manual preprocessing. For Task 2, NoRBERT performs consistently better compared to the model by *Kurtanović and Maalej* in both binary and multiclass classification settings. For multiclass classification specifically, NoRBERT achieves a weighted average F1-score of 87%. For Task 3, NoRBERT outperforms *Abad et al.*'s model without manual preprocessing, making it more adaptable for practical applications. Better performance is achieved with multiclass classifiers compared to binary classifiers across most classes. While BERT-large performs best overall, particularly on evenly distributed data, it encounters difficulties with highly imbalanced datasets. Despite these challenges, NoRBERT demonstrates strong performance in identifying underrepresented NFR subclasses, even in set-ups with small amounts of training data and in scenarios involving unseen projects. On Task 4 NoRBERT exceeds *Kurtanović and Maalej* by 15 percentage points on average.

NoRBERT demonstrates robust performance on unseen projects, which is crucial for practical applications, and performs especially well in scenarios with limited training data, thus reducing the need for extensive domain-specific data and making it feasible to apply the model across various projects without extensive retraining.

Li et al.[17] propose DBGAT, which leverages a graph neural network, Graph Attention Network (GAT), combined with BERT embeddings, to automate requirements classification while also achieving a better understanding of the relationships within the text, leading to improved classification outcomes.

The model is designed to utilize dependency parse trees to construct dependency graphs that save the syntactic features and dependencies in requirement sentences as the edge information of the graph. BERT is employed to initialize the node embeddings within the graphs, enabling the model to incorporate rich contextual features. The graphs are then processed using a GAT to capture the implicit structure feature and syntactic feature of requirements and learn their network embedding. Based on the initialized node embeddings by BERT, the information from the forward and backward neighborhoods of nodes in the graph are aggregated to construct bidirectional node embedding. After that and based on the learned node embedding, the graph embedding capturing the information of the entire graph is obtained. Finally, the maximum pooling method is used to aggregate the node vector output of the GAT model, then passed through a multi-layer perceptron for classification, with a softmax function providing the probability distribution over the requirement candidates subclasses.

Again, the three tasks under evaluation are classification into functional and non-functional requirements on the PROMISE dataset (tested using a 10-fold cross-validation setting); multi-class classification of the four most frequently occurring NFR subclasses on the PROMISE dataset combined with the heterogeneous Concordia RE dataset to test the generalization capability of the model (10-fold cross-validation and project-level cross-validation are used); finally, multi-class classification of all NFR subclasses on the PROMISE dataset using 10-fold cross-validation.

DBGAT is compared to existing state-of-the-art approaches, including ML methods like RF and NB, NoRBERT, and graph neural network-based models alone. The results compare the weighted average F1-scores and show that DBGAT outperforms NoRBERT and, in general, all methods that do not require manual pre-processing, especially on the NFR subclasses classification task and the all NFR subclasses classification task, while for FR/NFR classification the performance is similar. In Task 2, the model's performance on unseen projects shows a decrease of only 5%, demonstrating robust generalization.

The study concludes that integrating GNNs with BERT embeddings offers a powerful approach

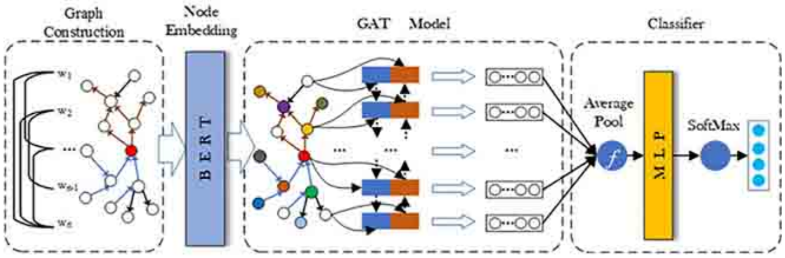


Figure 3.10: DBGAT structure and components. Firstly, dependency parse trees are used to build the graph structure of requirements sentences. Then BERT initializes the nodes/edges in the graph, and GAT captures more node features through training. Finally, use the classifier for requirements classification.

to requirements classification, particularly for capturing the complex syntactic and contextual features of non-functional requirements. DBGAT not only improves classification accuracy but also demonstrates robust generalization across different projects when applied to unseen datasets, which makes it a valuable tool for automating requirements classification in diverse software engineering contexts.

Another approach combining contextual word embeddings with advanced network architectures for automatic classification of NFRs is NFRNet by *Li and Nong*[18]. The model is designed as a two-part network. The first involves an improved BERT word embedding model that uses N-gram masking to learn the context representation of requirement descriptions. In contrast to the original BERT model, which masks 15% of words using a character-level approach, N-gram masking uses a variable-length N-character sliding window for masking, which enables the model to consider both the character and the word or phrase sequence features, and learn the intrinsic dependencies and interrelationships of continuous text for better representation learning during pre-training. The context feature vector calculation involves computing token, segment, and position embeddings for a requirement description. Then at each iteration, these embeddings are summed and the result of the previous iteration is spliced to obtain the feature vector C. Feature vector C is fed into a multi-layer transformer mechanism where each transformer node takes the output matrix of all nodes in the previous layer as input, and the final context feature vector E is calculated. The second part is a Bi-LSTM classification network that captures the bidirectional context information from the requirement descriptions. The final layer is a softmax classifier that converts the outputs of the network into probabilities, allowing to predict the category of each requirement.

During the training process of the deep neural network, the model uses a regularization technique called multi-sample dropout that improves the generalization ability of the model, allows to alleviate the increase in training time present with traditional dropout, reduces the number of iterations needed for training, and achieve lower error rates. Moreover, to ensure the fairness of the experiment, all the evaluated classification models adopt the same early stopping criterion and hyperparameters selection strategy by grid search. NFRNet is trained using an extension of the existing PROMISE dataset, the SOFTWARE NFR dataset, including a much

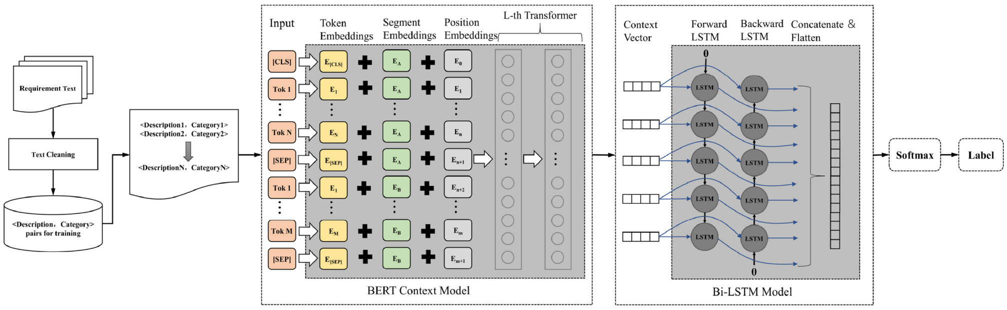


Figure 3.11: Overview of NFRNET.

larger number of NFR categories and requirement description sentences, and is evaluated on the task of NFR classification against 17 benchmark models including NoRBERT and DBGAT, along with various CNN, RNN, and transformer-based architectures.

The results of NFRs categories prediction on the SOFTWARE NFR dataset reveal that the NFRNet model outperforms all benchmark models in terms of precision, recall, and F1-score, including NoRBERT and DBGAT thanks to the combination of BERT's N-gram masking word embedding model and the Bi-LSTM model. These results thus show that the proposed approach can provide valuable assistance for analysts and developers, particularly in real-world applications, by reducing the need for manual work and alleviating the risk of introducing mistakes.

Tikayat et al.[19] propose the domain-specific aeroBERT-Classifier to accurately classify aerospace requirements, facilitating their integration into Model-Based Systems engineering tools and workflows. First, an annotated aerospace corpus is created with requirements organized into design, functional, and performance categories. After data preprocessing, which includes converting paragraphs into individual requirements, replacing symbols, and annotating them with appropriate labels (e.g., design, functional, performance), the sequences are tokenized to generate the proper input format for BERT. The model is then fine-tuned on the labeled dataset, with the token embeddings fed into the pre-trained BERT model (e.g., BERT-base-uncased or BERT-large-cased). The representation for the [CLS] token, containing the aggregate sequence representation, is passed through a pooler layer with a tanh activation function, followed by a linear classification layer to estimate class probabilities and assign the requirement to the category with the highest probability.

Of all the evaluated BERT variants, uncased models outperformed cased ones, indicating that casing is less crucial for text classification tasks like requirements classification compared to other tasks such as NER. Both aeroBERT models using BERT-large-uncased and BERT-base-uncased achieved the same overall average F1-score of 83%, suggesting high accuracy and reliability, and generalized well across categories despite the imbalance in the dataset. Given that the BASE-uncased model achieves similar performance but with less training time, it is preferred.

Moreover, the models' performance remains consistent across multiple iterations of training and testing, with balanced precision and recall and a high F1-score, further supporting the model's robustness. Since the difference between the training and test performance is low despite the small size of the dataset, it is expected that the model will generalize well to unseen requirements belonging to the categories under consideration. However, for certain ambiguous or complex requirements that were consistently misclassified, a human-in-the-loop approach may be necessary to ensure proper classification and handle edge cases.

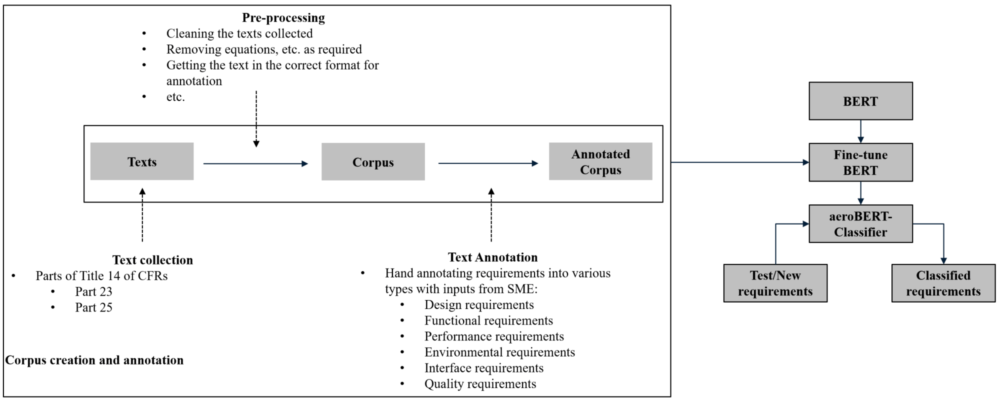


Figure 3.12: Methodology for obtaining aeroBERT-Classifier.

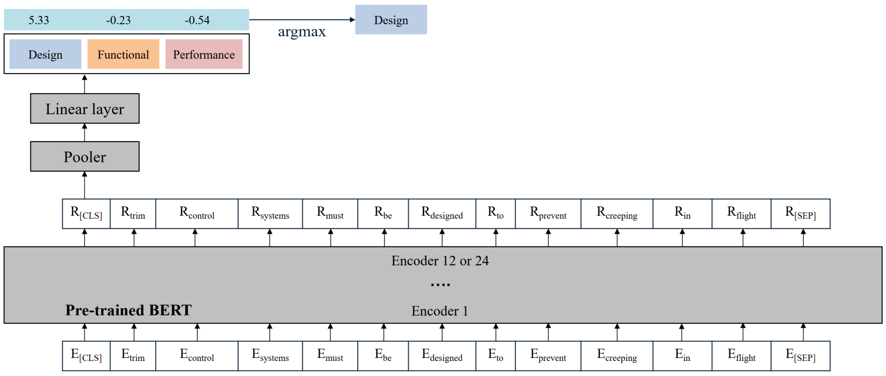


Figure 3.13: Detailed methodology used for the full-fine-tuning of BERT (base and large) LM is shown here. E_{name} represents the embedding for that particular WordPiece token, which is a combination of position, segment, and token embeddings. R_{name} is the representation for every token after it goes through the BERT model. Only $R_{[CLS]}$ is used for requirement classification since its hidden state contains the aggregate sequence representation.

AeroBERT-Classifier is also compared to other text classification LMs and shows superior performance in classifying aerospace requirements compared to GPT-2, Bi-LSTM with GloVe, and the ZSL classifier bart-large-mnli [56] without prior training/fine-tuning. AeroBERT-Classifier achieves an F1-score of 83%, which further underscores its reliability and effectiveness compared to lower-performing alternatives. While GPT-2 and Bi-LSTM provided competitive results, the zero-shot bart-large-mnli model performed significantly worse, especially when dealing with specialized and structured aerospace requirements. This highlights the importance of training/fine-tuning on even a small domain-specific labeled dataset for effective transfer learning and thus better generalization. Zero-shot classification, on the other hand, while useful on general tasks such as sentiment analysis and classification of general-purpose text, degrades performance when used for specialized and structured texts such as aerospace requirements.

Tikayat et al.[20] develop an agile methodology that leverages LLMs to standardize requirements, converting them from natural language into machine-readable formats that can be integrated into MBSE environments, and facilitating their validation and analysis earlier in the development process.

The approach consists of a two-fold strategy, first using LLMs to extract relevant information from NL requirements and organize it into a structured table for future automated analysis and model integration; second, identifying patterns to create standardized boilerplate templates that can help engineers write consistent and clear requirements, reducing ambiguity and improving readability.

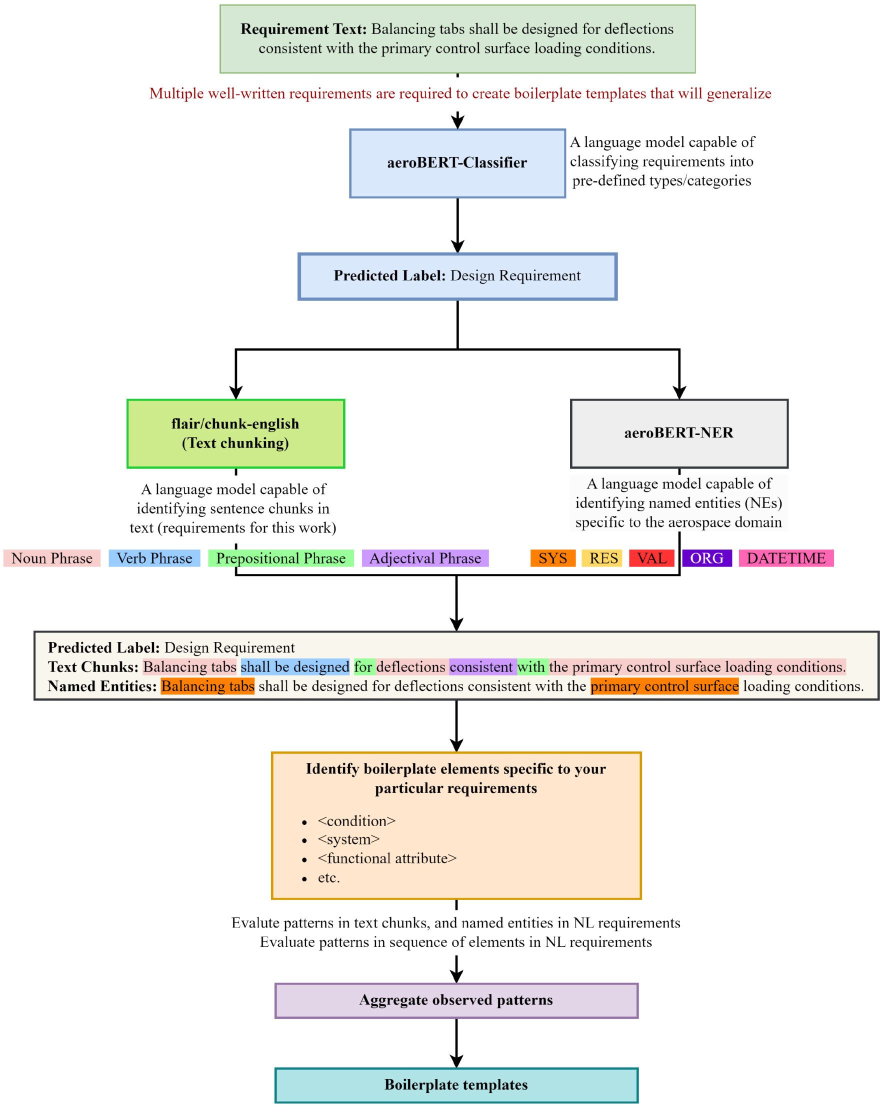


Figure 3.14: Flowchart showcasing the creation of boilerplate templates using aeroBERT-Classifier, aeroBERT-NER, and flair/chunk-english.

The approach employs aeroBERT-Classifier for requirements classification into design, functional, or performance categories; aeroBERT-NER [19], to identify named entities within requirements; flair/chunk-english [57] to identify sentence chunks within requirements, aiding in the recognition of linguistic patterns. aeroBERT-Classifier and aeroBERT-NER are fine-tuned on annotated aerospace texts, while no specific fine-tuning was performed on flair/chunk-english.

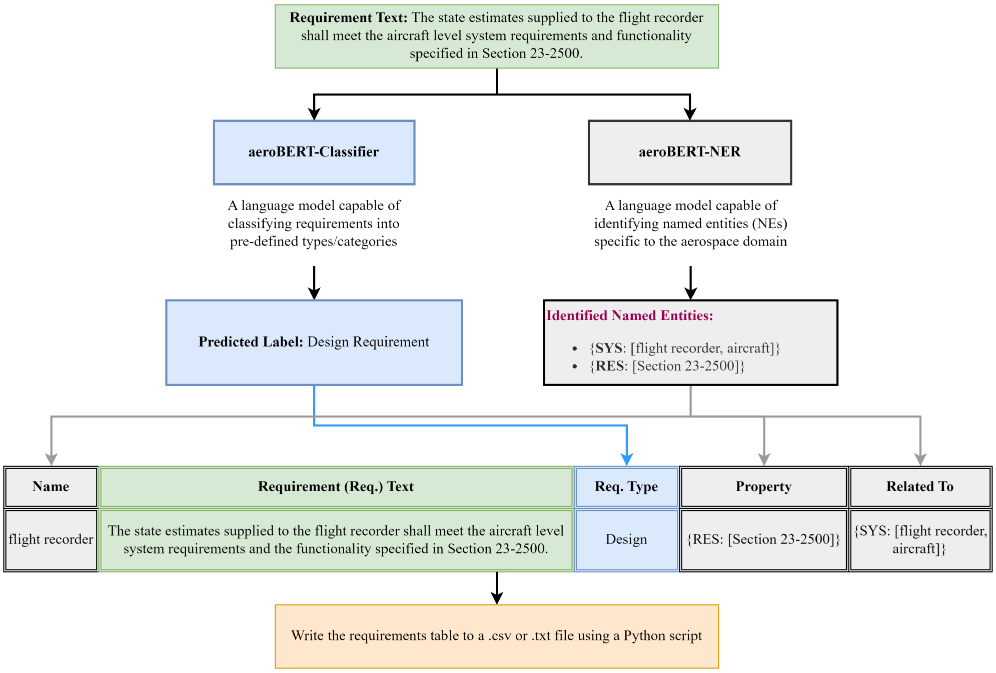


Figure 3.15: Flowchart showcasing the creation of the requirements table for a specific requirement using aeroBERT-Classifier and aeroBERT-NER to populate various columns of the table.

First, requirements are classified into various categories, then sentence chunks and named entities are obtained, and boilerplate elements are identified based on detected patterns. The information extracted from requirements is stored in tabular format and used to develop boilerplate templates for different types of requirements. Each requirement is represented by a row, and its associated attributes or system model elements are captured in columns. The use of requirement tables not only helps in managing and monitoring requirements throughout the development process but also in identifying dependencies, conflicts, redundancies, and potential oversights. Compared to the dictionary-based approach by *Riesener et al.* [58] where there is a need for constant updating as the requirements evolve, this semi-automated process leveraging LLMs minimizes manual intervention and helps to streamline the integration of requirements into MBSE environments.

Two boilerplate templates were identified for design requirements, covering approximately 55% of the dataset, five for functional requirements, covering about 63%, and three for performance requirements, covering around 58%. These templates capture a significant portion of the dataset, demonstrating the effectiveness of the approach in standardizing natural language (NL) requirements. The identified boilerplates allow for the creation of new requirements that follow established patterns or for assessing the conformity of existing requirements. Care was taken to avoid overfitting by limiting the number of templates, ensuring their generalizability across a wide range of requirements. It is also worth mentioning that the flair/chunk-english model shows some limitations in correctly identifying aerospace-specific terms due to its general-purpose training, therefore when the text chunking and the NER models disagree, the results from aeroBERT-NER should take precedence, ensuring better handling of domain-specific terminology.

The methodology presented offers a valuable approach for standardizing requirements on a larger scale and at a faster pace, but is adaptable across various domains and can be expanded to incorporate additional requirement types and named entities through further fine-tuning of the models, ensuring broader applicability and flexibility without sacrificing accuracy.

Alhoshan et al.[21] explore zero-shot learning (ZSL) as a means to improve requirements classification without labeled training data and with inherent generalization ability on unseen projects, as it directly applies previously trained models to predict both seen and unseen classes. ZSL is evaluated on different classification tasks, with different language models and label configurations, using the full PROMISE NFR dataset and the SecReq dataset.

The study uses an embedding-based ZSL method where both the class labels and input text are represented as word sequences using word embeddings, and classification is performed by calculating the semantic similarity between the two. The contextual word embeddings are obtained through transformer-based language models, including both general-purpose and domain-specific models, such as Sentence-BERT, All-MiniLM-L12 (AllMini), Bert4RE[59], and BERTOverflow (SObert) [60]. The proposed embedding-based ZSL model does not use similarity thresholds but treats all text classification tasks as a multi-label classification task, where all labels are ranked based on their similarity scores. For binary or multi-class classification tasks, the label with the highest similarity score is chosen as the predicted label, for multi-label tasks the top-n labels are considered. It is therefore clear that the approach highly depends on the choice of labels and language models. The different label creation strategies explored include original labels derived from the original class names used in the dataset, expert-curated labels, and word-embedding generated labels, extracted and selected from word-embeddings learned from Wikipedia Computer Science text.

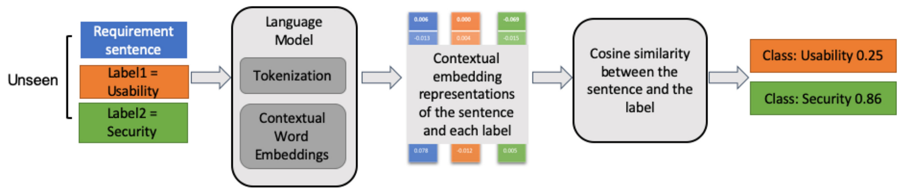


Figure 3.16: Illustration of the contextual word embedding-based ZSL approach.

For Task 1 -FR/NFR classification-, the best combination of model and label configuration is the domain-agnostic model sentence-BERT and the combination of expert-curated and original label, with a weighted F1 score of 66%, precision of 0.71, and recall of 0.66. Using expert-curated and original labels specifically helps in identifying NFRs through class names like Usability and Security. Notably, the model shows higher precision on NFRs and higher recall on FRs, despite FRs being less frequent in the dataset. This suggests that in a zero-shot setting, performance doesn't depend on the size of the dataset for each class as no actual learning is performed on the labeled data. Further improvements in classifying FRs might be achieved with a more project-specific labeling strategy.

For Task 2 -NFR classification-, three subtasks are performed: binary classification to identify whether a requirement belongs to a specific NFR category; a multi-class classification to assign each requirement to one NFR category from a set of classes; and a multi-label classification where each requirement is associated with a ranked list of NFR categories, checking if the correct one is among the top-k choices. The results are evaluated first focusing on the four largest NFR classes and second on all NFR classes. The results show that the approach performs better on binary classification tasks, and suggest that for well-represented NFR classes like security and usability, combining language models with word embeddings as labels is highly effective, and that performing binary classification on the largest NFR classes is preferable compared to classifying FRs vs. NFRs. However, the classifiers tend to perform better on the "Other" class than on the specific NFR class, indicating difficulty in correctly associating requirements with specific labels despite using extensive term sets. This issue might be mitigated by selecting more accurate terms or using terms directly extracted from the requirements. When considering all classes, the performance for the largest classes remained stable, nonetheless, the results gained are all biased toward the "Other" class, with the F1 score for the NFR class dropping below 0.50. This bias was less pronounced when focusing solely on the largest classes. Therefore, we conclude that in scenarios involving many different classes, binary classification leads to poor results with the current labeling strategies. Compared to binary classification, multi-class classification of NFRs yields significantly lower results, though acceptable for some classes like Security and Performance. The best performance was achieved using generic LMs with simpler labels, as opposed to the binary classification, as longer labels may cause overlapping meanings between classes, reducing performance. When all NFR classes are considered, only the Security class maintains acceptable performance, while other classes show poor results.

In multi-label classification of the four largest non-functional requirements (NFR) classes, considering the top two labels significantly improves performance compared to multi-class classification. There is no clear pattern regarding which language models perform best, as each seems suitable for different requirement types. Simple label configurations like original and expert-curated labels are generally the most effective for all classes except usability, where embedding-based labels work better, possibly due to a clearer characterization of US requirements. When extending the multi-label classification to all NFR classes and considering the top three results, performance remains high. Overall, zero-shot learning in a multi-label context appears to be effective even for NFRs belonging to many classes.

For Task 3, the best results are achieved with the AllMini model and expert-curated labels. However it cannot be concluded that generic LMs are better in this particular task, since the other generic model, Sbert, achieves the worst results. Sbert with word embedding generated labels though had the highest recall, making it better for identifying as many security-related items as possible, minimizing false negatives.

The study highlights that in the majority of the cases, generic LMs perform better than domain-specific LMs on requirements classification tasks under the ZSL approach. The choice of label configuration also significantly influences the model's performance. While more complex word-embedding generated labels yield better results in binary classification of NFRs, simple label configurations (using original class names or a combination of original and expert-curated labels) generally appear more effective.

Overall, the performance achieved by the best ZSL classifiers (that is, the best combinations of LM and label configurations) is still lower in comparison with supervised learning approaches, however, the approach presents crucial advantages as it is fully unsupervised, does not require any labeled dataset or training, and is thus more flexible, and easily adaptable to the evolution of classification schemes with unseen classes. Hence the authors conclude that zero-shot learning with careful selection of language models and label configurations is a viable and effective approach for requirements classification especially in scenarios where labeled data is scarce or unavailable and with large sets of non-mission-critical requirements, where some misclassification can be tolerated. Finally, to achieve the best results in ZSL for multi-class classification, a multi-label approach is recommended.

One limitation of the main requirements classification approaches is that they are conducted on the small-size PROMISE dataset. Moreover, since simply adding a neural network after a transformer-based model may result in abruptness and a low correlation between the input sequence and the target task, a model like NoRBERT [16] no longer appears as the most ideal technique.

To address current limitations in automated requirements classification and enable the auto-labeling of unseen requirements categories from large-scale datasets, *Luo et al.*[22] propose PRCBERT. The approach leverages a labeled dataset of requirement samples and flexible prompt templates to obtain a fine-tuned model. The fine-tuned model is then used for requirement inference (prediction) and finally combined with self-learning to perform zero-shot learning on the unseen requirements of a large-size dataset (NFR-SO) consisting of real-world software requirements from technical forums like StackOverflow built by the authors.

Specifically, the proposed approach called PRCBERT utilizes flexible prompt templates to convert one K-class classification problem into K binary classification problems. The key idea is to add a prompt text into the input sequence to enhance semantic fluency between the input and target task, bridging the semantics gap between general pre-trained language models and the classification task at hand. The prompt template used is “This is <requirement category name> Requirement".

The train set for fine-tuning is constructed by duplicating each requirement sample from a well-labeled dataset (such as PROMISE) into K samples -where K represents the number of label classes or requirement categories-, and then concatenating each duplicated sample with a different prompt template. The concatenated result is referred to as an assertion. After obtaining the final hidden state from the transformer-based model, PRCBERT uses a mean pooler strategy to compute the assertion representation, which is then passed to a two-class sigmoid layer for binary classification, to predict the validity of assertions. After getting the classification confidence, the training objective loss and the model's learnable parameters are iteratively calculated and updated through backpropagation, until PRCBERT convergences or a predefined epoch threshold is reached.

At the inference step, the duplicated assertions of a given input sample are input to the fine-tuned PRCBERT to get the predicted class label for each requirement sample, represented by the one with maximum classification confidence.

For the final step of automatically classifying unlabeled requirements, the fine-tuned PRCBERT model from the inference step is used to perform zero-shot classification on unseen requirements from another requirement dataset (NFR-SO) using a self-learning strategy. In this process, samples with higher confidence are selected to iteratively update PRCBERT: initially, a coarseSet is generated by randomly sampling from the dataset; this set is refined by filtering out samples whose confidence scores, as predicted by PRCBERT, fall below the threshold; PRCBERT then updates its parameters through gradient descent based on the refinedSet. This self-learning process continues iteratively until no further samples are filtered out. This strategy is adopted to boost the model's performance in zero-shot scenarios, improving the transfer learning and generalization ability for auto-labeling unseen requirements. Ultimately, the algorithm outputs the final predicted labels for the entire dataset.

In the experimental results on binary classification of FR/NFR and NFR Subclasses from PROMISE, PRCBERT consistently outperforms the baseline model NoRBERT achieving the

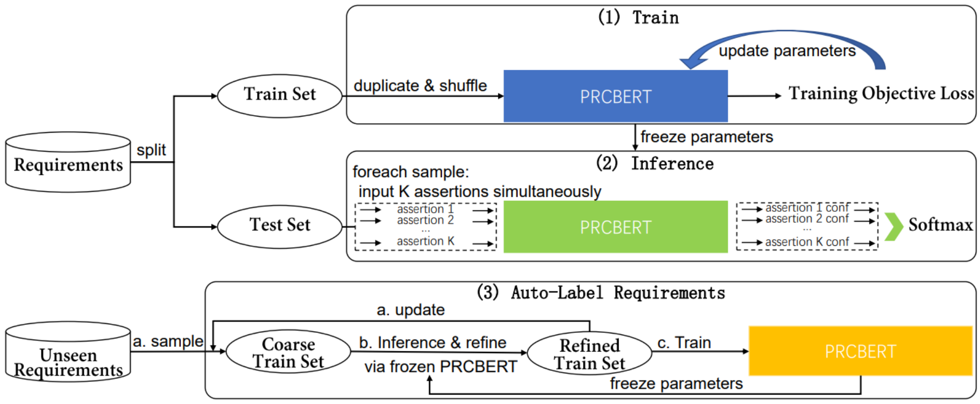


Figure 3.17: Overview of PRCBERT approach. The top part corresponds to the train (fine-tune) step and the inference step respectively; and the bottom part represents the proposed auto-labeling method (zero-shot + self-learning) for unseen requirements.

best results with the RoBERTa-large model and mean pooling strategy to extract sequence vector representations from the final hidden state of the transformer-based model. These results highlight PRCBERT's improved performance due to its flexible prompt templates and binary classification approach.

In classification on NFR-Review, PRCBERT outperforms NoRBERT and BERT-MLM among all train-set sizes. Trans_PRCBERT, the version of PRCBERT fine-tuned on PROMISE, and subsequently trained on NFR-Review or NFR-SO (rather than directly trained on NFR-Review or NFR-SO like PRCBERT) achieves the best performance as it can be directly applied on NFR-Review preserving the semantic understanding learned from PROMISE to a certain extent. This highlights PRCBERT's strong transferability and ability to generalize across different datasets.

In classification on NFR-SO, PRCBERT with flexible prompt templates achieves the maximum weighted F1 score of 86.22%, outperforming NoRBERT and BERT-MLM among all train-set sizes. Trans_PRCBERT which is transferred from PROMISE to NFR-SO, achieves the best results compared to NoRBERT, BERT-MLM, and PRCBERT models that use random initialization for downstream classification at the beginning. Trans_PRCBERT is able to auto-label the new NFR-SO dataset without any adaptation of the classification neural network which is added behind the pre-trained transformer-based language model, showing high generalization and transferability.

Concerning auto-labeling unseen requirements, NFR-REVIEW and NFR-SO are de-labeled and used as input for the auto-label algorithm. Trans_PRCBERT with a self-learning strategy exhibits significantly better performance than zero-shot performance with PRCBERT and with Trans_PRCBERT.

PRCBERT significantly improves the accuracy and robustness of software requirement classification. by leveraging pre-trained transformer models and flexible prompt templates. Its self-learning strategy enhances zero-shot classification performance, making it highly effective across both small (PROMISE) and large-scale (NFR-SO) datasets. The approach consistently outperforms existing models, demonstrating superior transferability and applicability to unseen requirements.

El-Hajjami et al.[23] present an empirical evaluation of different AI techniques for requirements classification into functional and non-functional requirements, and other related categories. The study employs five public datasets (PROMISE, Dronology, ReqView, Leeds Library, and WASP) and uses a range of models for requirements classification into categories based on whether they include functional aspects (IsFunctional), quality aspects (IsQuality), are only functional (OnlyFunctional), or only quality (OnlyQuality).

The evaluated models include SVM, LSTM, and GPT through the ChatGPT interface. SVM utilizes traditional text n-grams and Part-of-Speech (POS) n-grams features. The LSTM architecture has an embedding layer to convert words into numerical vectors, a spatial dropout layer for regularization to prevent overfitting, the LSTM layer that processes text sequences and captures word dependencies, and a dense layer with a sigmoid activation function to output probability scores indicating the likelihood of a requirement being in the positive class. Finally, both GPT-3.5-turbo and GPT-4 were tested in zero-shot and few-shot settings using the OpenAI API with default parameter settings except for temperature (set to 0 to minimize randomness and enhance reproducibility). In the zero-shot prompting approach, the models perform tasks relying solely on their pre-trained knowledge and the immediate context of the prompt, testing the model's ability to generalize from its training data. In this study, the zero-shot prompts are designed with three key principles: they include all necessary details for relevant answers, instruct the model to act as a software requirements expert, and use wording from *Dalpiaz et al.*. Few-shot learning adopts the same principles but also leverages examples of the inputs and their corresponding outputs. These examples are carefully selected from the PROMISE training set in order to ensure a balanced class representation that exposes the models to a diverse range of scenarios and provides a comprehensive understanding of the classification nuances.

SVM and LSTM are trained on 75% of the PROMISE dataset and all models are evaluated on the same global evaluation dataset, which includes the remaining 25% of the PROMISE dataset and the full datasets from Dronology, ReqView, Leeds Library, and WASP.

The study uses F__β scores as evaluation metrics to assess the models' performance. Indeed, while the F1-score assumes precision and recall are equally important, F__β emphasizes recall, which is more important in the context of the study due to the greater feasibility of manually rejecting a false positive identified by the model rather than manually finding a true positive in the input. The value β is chosen based on the inverse frequency of the target class in the dataset, reflecting the importance of correctly identifying positive cases in imbalanced datasets.

GPT-3.5 in the few-shot setting achieves the highest F__β score for "IsFunctional" and "OnlyQuality" classifications (89.9% and 73.2% respectively), while GPT-4 in the zero-shot setting leads in the "OnlyFunctional" classification (87.2%) . LSTM performs best for the "IsQuality" classification with 70.8% F_β score. However, it performs particularly well on the PROMISE test set, likely due to being trained on similar data, while it is not consistent across the other test sets, showing limited generalization ability. GPT-3.5 generally outperforms GPT-4 in most scenarios except for the "OnlyFunctional" classification, which means that GPT-3.5 is preferred in most cases due to achieving better performance with minor cost, while the use of GPT-4 might be justified for specific tasks requiring high accuracy in functional requirements classification. Regarding the zero-shot/few-shot comparison, the few-shot setting brings significant improvements, especially in contexts where the zero-shot performance is weak, such as the "OnlyFunctional" and "OnlyQuality" classifications using GPT-3.5.

The authors conclude that there is no single best technique, and the optimal model depends on the specific classification task and the considered class to be identified. Future Work could be comparing ChatGPT with other LLMs like Llama-2 and Mistral for various RE tasks and developing high-quality benchmark datasets for comprehensive evaluations.

Xanthopoulou et al.[24] explore and compare various techniques for requirements classification ranging from simple ML and NLP methods to advanced transformer-based approaches including models commonly left out in previous works. In particular, the tasks encompass FR/NFR binary classification and multi-class classification of NFRs into the four largest categories - security (SE), usability (US), operability (O), and performance (PE)-, while excluding those with an excessively limited number of records.

As a first step, the authors construct a large, augmented dataset -suitable for both binary and multi-class requirement classification tasks-, to encourage greater generalizability of results, as the models are validated on a larger and more diverse dataset than previous works. The dataset is created by merging and labeling data from various publicly available software requirements datasets like PROMISE, PURE, SecReq, and Kaggle. Next, the data is pre-processed and the various classification models are built. For traditional ML models, the RF algorithm is used as it is found to be the best-performing model, both using BoW and Word2Vec embeddings; the other models are BERT and its variants DistilBERT and RoBERTa, GPT-2, BART, and T5.

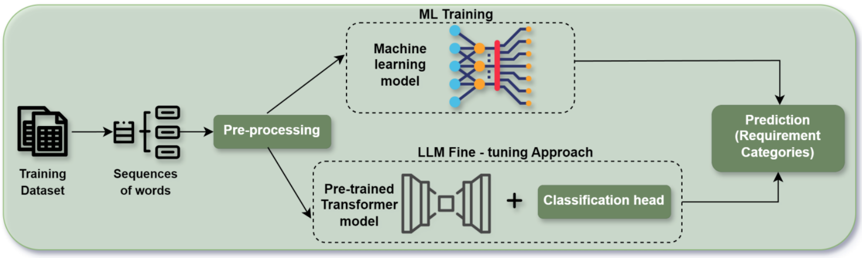


Figure 3.18: Methodology followed.

75% of the dataset is used for training the models, while the remaining 25% is split into validation and test sets. The validation set is used for tuning and hyperparameter selection, whereas the test set is used for evaluation on unseen data. All models achieve a satisfactory F1-score, but transformer-based models generally show a better predictive performance than ML methods leveraging simpler NLP techniques. In line with previous works, BERT and its variants achieve an F1-score higher than 90% both in binary and multi-class classification, while BART and GPT-2 demonstrate higher accuracy, which, for multi-class classification, indicates the models' ability to efficiently recognize more types of NFRs beside the majority class.

The results reveal that ML models built on BoW perform quite well in binary classification, achieving an F1-score of 90%, comparable to BERT-based models. This shows that functional and non-functional requirements do not generally overlap and are characterized by distinct words that are not found in both types of requirements. BoW also outperforms Word2Vec, indicating that the ML classifiers likely prioritize the presence of specific words over their syntactical arrangement within the sentences.

However, techniques like Word2Vec and BoW generate static, global word vectors, whereas transformer-based models have context-aware embedding vectors, can learn long-term dependencies between words, and also handle the out-of-vocabulary issue thanks to their use of sub-word tokenization. Therefore these models achieve better results.

In particular, BART emerges as the best model for binary classification with an F1-score of 94%, while GPT-2 excels in multi-class classification with an F1-score of 95%. These findings suggest that not only encoder-only models like BERT, but also decoder-only and encoder-decoder transformer-based architectures are viable solutions for requirements classification, providing similar or even better results, and are thus worth further exploration.

Requirements classification focuses on the distinction between functional and non-functional requirements and is also linked to their conversion conversion into structured or semi-structured formats. Methods based on traditional ML and NLP techniques suffer from limitations due to poor generalization on unseen classes and projects and scarcity of task-specific labeled datasets for training; reliance on manual pre-processing; lack of structural and syntactic text understanding. RQ2 examines recent advancements using large language models like BERT, assessing their effectiveness in requirements classification.

Hey et al.[16] present the NoRBERT model leveraging BERT and its transfer learning capability for classifying software requirements on the PROMISE NFR dataset and a relabeled version of it by *Dalpiaz et al* [52] in order to improve generalization. NoRBERT fine-tunes the base and large versions of BERT on a small amount of labeled data for the target task and is evaluated against state-of-the-art approaches in various settings, including a 75%-25% train-test split, stratified 10-fold cross-validation, and project-specific folding strategies. On FR/NFR binary classification the approach based on manual preprocessing and NB classifier by *Abad et al.* [54] achieves the best results but NoRBERT has the advantage of not requiring manual preprocessing while also performing similarly to approaches by *Kurtanović and Maalej* [53] (using automated feature selection and SVM) and *Dekhtyar and Fong* [55] (using word embeddings and a CNN). For binary and multiclass classification of the four most frequent NFR subclasses, NoRBERT performs consistently better than *Kurtanović and Maalej*'s model. For multiclass classification of all NFR classes, NoRBERT outperforms *Abad et al.* model without manual preprocessing. In all settings, the multiclass classifier and the large models outperform the respective binary classifier, demonstrating the model's strength, particularly on evenly distributed data and difficulties with highly imbalanced datasets. NoRBERT outperforms both *Kurtanović and Maalej* and *Dalpiaz et al.*, in all evaluation settings, on binary classification of functional and quality aspects. Therefore, overall, NoRBERT enhances requirements classification even with small amounts of labeled training data and on unseen projects.

Li et al.[17] present DBGAT, a model that integrates BERT embeddings with a Graph Attention Network (GAT) to automate requirements classification. The model leverages dependency parse trees to build a graph representation of requirements sentences; then BERT initializes node embeddings representing requirements information and the GAT model mines the implicit structure feature and syntactic feature of requirements, learning bidirectional node embeddings and graph embedding. Finally, a multi-layer perceptron is used for classification, with a softmax function providing the probability distribution over the requirements candidate subclasses. DBGAT outperforms all baselines that do not require manual pre-processing on the four most frequent NFR subclasses classification and all NFR subclasses classification, and performs similarly on the FR/NFR classification task. The model also shows robust generalization to unseen projects

NFRNet by *Li and Nong*[18] addresses the automatic classification of NFRs, combining an improved BERT word embedding model based on N-gram masking and a bidirectional-LSTM classification network with multi-sample dropout. In contrast to the original BERT model, which masks 15% of words using a character-level approach, N-gram masking uses a variable-length N-character sliding window for masking, which enables the model to consider both the character and the word or phrase sequence features. The multi-sample dropout regularization technique used during the training process of the deep neural network, improves the generalization ability of the model, reducing overfitting, while also alleviating the increase in training time of traditional dropout and reducing the number of iterations needed for training. NFRNet is trained and evaluated on an extended version of the PROMISE dataset against 17 models including NoRBERT and DBGAT, along with various CNN, RNN, and transformer-based architectures. NFRNet outperforms all benchmark models in terms of precision, recall, and F1-score, thanks to the combination of BERT's N-gram masking word embedding model and the Bi-LSTM model.

Tikayat et al.[19] propose a domain-specific model called aeroBERT-Classifier, designed to classify aerospace requirements and facilitate their integration into Model-Based Systems Engineering (MBSE) tools. An annotated aerospace requirements corpus is first created, data is preprocessed, and then fed into BERT models to estimate class probabilities and assign requirements labels. Among BERT's tested variants, uncased models outperform cased ones, and since BASE and LARGE uncased BERT achieve the same F1-score, the BASE model is preferred due to shorter training time. Despite the imbalanced dataset, the model demonstrated robust performance across multiple iterations, although a human-in-the-loop approach may be needed for requirements that are consistently misclassified across the iterations. Compared to other text classification LMs, including GPT-2, Bi-LSTM using GloVe, and the zero-shot bart-large-mnli model, aeroBERT-Classifier shows superior performance, especially with respect to bart-large-mnli, underscoring the importance of fine-tuning on domain-specific labeled data, even in small quantity, for improved generalization in tasks involving specialized texts like aerospace.

Tikayat et al.[20] also present an agile methodology leveraging LLMs to standardize NL requirements for aerospace systems, improving integration into MBSE environments. The approach involves first extracting information from NL requirements using LLMs and organizing it into a structured table; second, identifying linguistic patterns to generate standardized boilerplate templates. AeroBERT-Classifier is used for requirements classification, while aeroBERT-NER identifies named entities, and flair/chunk-english extracts sentence chunks. The methodology generates respectively two, five and three templates for the three requirement categories (design, functional, or performance), covering significant portions of the dataset while also avoiding overfitting. Moreover, since the text chunking model is not fine-tuned and thus shows some limitations in identifying aerospace-specific terms, the results from aeroBERT-NER take precedence, ensuring better handling of domain-specific terminology. Overall, the identified boilerplates allow for the creation of new requirements that follow established patterns or for assessing the conformity of existing requirements helping to ensure consistency and streamline requirement validation and integration, offering a scalable solution that is also adaptable to other domains.

Alhoshan et al.[21] use zero-shot learning (ZSL) to improve requirements classification without labeled training data and with inherent generalization ability on unseen projects. The embedding-based ZSL method used performs classification by calculating the semantic similarity between the word embeddings of both the class labels and input text, obtained through transformer-based language models. ZSL is evaluated on FR/NFR classification; binary, multi-class and multi-label classification over NFRs; security vs non-security requirements classification. Different language models, both general-purpose and domain-specific, are tested, along with different label configurations, using the full PROMISE NFR dataset and the SecReq dataset. The study highlights that in the majority of the cases, generic LMs perform better than domain-specific LM and that, while more complex word-embedding generated labels yield better results in binary classification of NFRs, simple label configurations (using original class names or a combination of original and expert-curated labels) generally appear more effective. Multi-label classification achieves the best performance when using ZSL for multi-class classification. While the best ZSL classifiers still perform lower in comparison with supervised learning approaches, the approach has the advantage of being fully unsupervised and easily adaptable to unseen classes.

PRCBERT by *Luo et al.*[22] aims to address limitations of traditional requirements classification methods relying on the outdated and small PROMISE dataset and older NLP techniques and enhance auto-labeling of unseen requirements from large-scale datasets. The approach fine-tunes a BERT-based model leveraging a labeled dataset of requirement samples, each of which is duplicated K times (as the number of requirement categories) and concatenated with a different prompt template to achieve higher semantic fluency between input and target task, transforming one K-class classification task into K binary ones. The fine-tuned model is then used for requirement inference (prediction), and finally combined with self-learning to perform zero-shot learning on unseen requirements from a large-size dataset (NFR-SO). PRCBERT, directly trained on NFR-Review or NFR-SO, performs moderately better than NoRBERT and MLM-BERT (BERT with the standard prompt template) in binary classification of F/NF requirements and classification of NF subclasses from PROMISE. Trans_PRCBERT (PRCBERT fine-tuned on PROMISE and subsequently trained on NFR-Review and NFR-SO) performs best in classification on NFR-Review and NFR-SO and finally, achieves satisfactory performance in the task of auto-labeling unseen requirements performed on the de-labeled NFR-Review and NFR-SO datasets, especially when coupled with a self-learning strategy.

El-Hajjami1 [23] investigate the effectiveness of SVM, LSTM, and GPT models version GPT-3.5 and GPT-4 for classifying software requirements, using five public datasets. The study categorizes requirements based on whether they include functional or quality aspects. The key findings are that no single technique outperforms others in all scenarios, but the optimal model depends on the specific classification task and the considered classes. LSTM is the best-performing model for quality-related requirements classification, but particularly on the PROMISE test set; it struggles with generalization on the other test sets, likely because they are less similar to the data used during model training. GPT-3.5 in a few-shot setting leads in the "IsFunctional" and "OnlyQuality" classification, surpassing GPT-4 except for "OnlyFunctional" requirements. This makes it a more cost-effective option except when high accuracy is requested on functional requirements classification. Few-shot learning significantly boosts performance, especially in cases where zero-shot learning results are weak.

Xanthopoulou et al.[24] compare the RF algorithm using BoW and Word2Vec embeddings to transformer-based models including BERT and its variants DistilBERT and RoBERTa, GPT-2, BART, and T5, on a large augmented dataset combining publicly available sources (PROMISE, PURE, SecReq, Kaggle). While all models achieved satisfactory F1-scores, transformer-based models generally outperform ML methods. Notably, RF with BoW performs similarly to BERT-based models in binary classification, suggesting that distinguishing functional from non-functional requirements are often characterized by distinct words that are not found in both types of requirements. Among transformers, BART and GPT-2 achieve higher or comparable accuracy than BERT, with BART achieving the best results in FR/NFR binary classification, and GPT-2 for multi-class classification of non-functional requirements. Overall, the findings indicate that transformer architectures—including encoder-only, decoder-only, and encoder-decoder models—are viable solutions for requirements classification.

In conclusion, LLMs have significantly advanced requirements classification, with BERT-based models excelling in both binary and multiclass tasks, especially for non-functional requirements. Decoder-only models like GPT are found to perform well in few-shot learning, particularly for functional requirements, and encoder-decoder models such as BART and T5 prove effective in multi-class classification. These models outperform traditional ML and NLP approaches, offering improved generalization, automation, and adaptability across complex tasks and domains while also reducing the need for manual preprocessing.

3.2.3 RQ3: How can LLMs be leveraged for enhancing requirements completeness and managing ambiguity?

Due to systems and software requirements being commonly articulated in natural language, they are very often prone to incompleteness, ambiguity or contradiction. Traditional methods to address these issues often rely on rule-based systems and classical machine learning techniques, struggling with scalability and semantic understanding. In recent years, pre-trained language models like BERT and its variants have been evaluated across various ambiguity- or completeness-related tasks and have introduced significant improvements.

One very common type of ambiguity in NL requirements is anaphoric ambiguity, which occurs when a pronoun or other referring expression can point to more than one possible antecedent, leading to misunderstandings.

Existing solutions for handling anaphoric ambiguity usually focus on resolution of ambiguities and treat the task as part of coreference resolution [25]. This approach identifies references to the same entity within a text as an intermediate step before performing more advanced NLP techniques. These techniques include syntactic and semantic approaches (focused on designing ML features based on grammatical structure and word meanings in sentences), and neural-network-based approaches (where anaphora resolution is often interpreted as a question-answering task). However, RE prioritizes ambiguity detection, for which current coreference resolvers are not designed specifically. Moreover, another limitation in existing approaches is that they are trained on generic corpora and evaluated on small datasets or single domains.

Ezzini et al.[25] aim to address these limitations by developing a practical and efficient solution for handling anaphoric ambiguity. This primarily involves detecting instances of anaphoric ambiguity and secondly resolving them when the risk of ambiguity is low enough. In other words, this means deciding whether a given pronoun occurrence is ambiguous or unambiguous in its context, and identifying the most likely antecedent for the pronoun, given a candidate set. To achieve this goal, the authors conduct an empirical investigation of multiple solution strategies leveraging a relatively large RE dataset covering eight different application domains. The evaluation explores several aspects, including the use of hand-crafted language features, word embeddings, or a combination of both for classification; the viability of pre-trained language models like BERT as a replacement for more traditional techniques; and, finally, the effectiveness of combining existing, often generic, NLP tools for specific tasks in requirements engineering.

The alternative solutions proposed include six approaches. The first two are $SpanBERT_{NLP}$, fine-tuned on the CoNLL2011 dataset comprising annotations relevant to anaphoric ambiguity analysis, and $SpanBERT_{RE}$, with intermediate fine-tuning on CoNLL2011 and further fine-tuning on a subset of DAMIR, an NL requirements dataset. This dataset is created by the authors by collecting 22 industrial requirements specifications from eight different application domains. Two other approaches are based on supervised ML methods, trained on either language features (ML_{LF}) from the existing NLP and RE literature on anaphora resolution or feature embeddings (ML_{FE}) extracted from BERT and SBERT. On the other hand, $ML_{ensemble}$ combines the results of both these ML approaches. For these solutions, the authors experiment with a range of ML algorithms and feature extraction techniques and find the best-performing configuration of the two. The last proposed solution is based on existing NLP coreference resolvers.

The evaluation is conducted on a portion of DAMIR and ReqEval, another RE dataset on anaphoric ambiguity, borrowed from literature, and the metrics used are precision, recall, and F_β-score for ambiguity detection, and success rate for anaphora resolution (the ratio of correctly

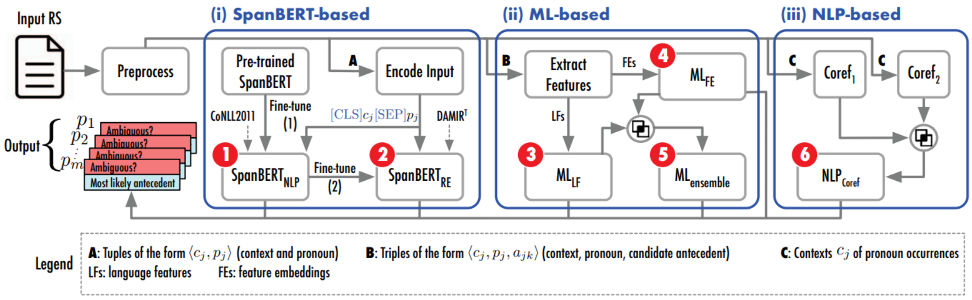


Figure 3.19: Overview of Solution Alternatives (marked 1 to 6)

resolved pronoun occurrences to the total number of pronoun occurrences labeled as unambiguous in the ground truth). Among the examined approaches, the most accurate and effective at detecting ambiguity is $ML_{ensemble}$, combining predictions from ML_{LF} and ML_{FE}, with a recall of 100% and an average precision of approximately 60%. The reason probably resides in these solutions being explicitly trained to distinguish ambiguous and unambiguous pronoun occurrences. For anaphora resolution instead, $SpanBERT_{RE}$ outperforms other solutions, emerging as the most accurate with an average success rate of around 98%. Overall, these results show that different approaches are better for different tasks, so a hybrid solution combining the two best-performing ones for ambiguity detection and resolution respectively, would be the optimal choice. The authors also highlight the importance on performance of the intermediate fine-tuning step of $SpanBERT_{RE}$, before further fine-tuning with specific requirements engineering data. Given the limited availability of specialized datasets for RE tasks, intermediate fine-tuning with additional datasets might be very valuable if not necessary for improving the performance of requirements automation solutions using pre-trained language models. Therefore, RE researchers might need to seek out supplementary datasets from other fields, such as NLP, to fully leverage the potential of these models.

Jafari et al.[26] investigate the effectiveness of various techniques for the two-fold task of detecting and resolving anaphoric ambiguity in requirements texts, including question-answering (QA) and text-to-text transformer models, and propose two models for anaphoric resolution based on pre-trained language models DeBERTa and T5. The analysis involves three datasets, the ReqEval and DAMIR requirements datasets, and the CoNLL2011 general-purpose pronoun resolution dataset. The models examined for anaphoric ambiguity detection include BERT and its variants RoBERTa, DistilBERT, and DeBERTaV3, fine-tuned on the datasets; the majority classifier and SVM classifier on top of feature extraction methods such as TF-IDF, GloVe word embeddings, and SBERT as baselines.

For ambiguity resolution, the authors compare two approaches. Encoder-based extractive QA uses BERT-based models BERT, RoBERTa, SpanBERT, and DistilBERT to extract the correct antecedent from the text, treating the task as a question-answering problem. Similarly to a token classifier, the model identifies the start and end of a possible answer by classifying the tokens. The encoder-decoder-based text-to-text approach employs T5-small and T5-base models designed for text generation. In this case, the model generates the most likely antecedent based on the context rather than extracting it, differently from BERT-based encoder-only models which can only output class labels to indicate spans and phrases.

The experiment results demonstrate that for anaphoric ambiguity detection, BERT-based models outperform baselines in terms of macro F1 scores, and particularly DeBERTaV3 achieves the overall best performance. However, due to the limited number of supervised datasets utilized, the general performance in this task is still not very satisfactory, which suggests that the models may benefit from more thorough fine-tuning and techniques such as few-shot learning and contrastive learning.

For the anaphoric resolution task, even though in some cases the T5 model achieves comparable or even better performance, extractive QA models overall outperform text-to-text models, especially RoBERTa model pre-trained on the SQuAD 2.0 dataset for downstream QA tasks. Compared to the models proposed by [25], they achieve similar results on the DAMIR dataset, but lower accuracy values on the ReqEval dataset. Possible explanations for the low performance of text-to-text models are the phenomenon of hallucination and the use of T5 small and base models due to limited computing resources, whereas bigger models might be improving the results.

The authors also conduct experiments on the three datasets using generative decoder-only models GPT-3.5 and GPT-4 in comparison with the best-performing resolution model RoBERTa. The evaluation reveals that encoder-only RoBERTa-based QA model achieves superior performance than the decoder-only generative GPT models on all datasets. Indeed, GPT models, designed to excel at zero-shot generalization across various tasks, often generate unintended or incorrect outputs unrelated to the input instructions, plus, it is unclear whether they actually do better than smaller, more cost-efficient models like BERT with task-specific fine-tuning, and finally, due to their large number of parameters, these models may not be feasible in environments with limited computational resources.

To conclude, while BERT-based models excel in both detection and resolution tasks, generative models like T5 and GPT show potential but require further refinement to handle the complexities of requirements engineering effectively.

Another important aspect in requirements analysis is the detection and resolution of conflicts among requirements. While traditional methods to approach the task generally require a significant amount of labeled, task-specific training data and manual intervention, *Helmeczi et al.*[27] propose a prompt-based few-shot learning approach for detecting conflicts in software requirement specifications, by classifying requirement pairs as either conflicting, duplicate, or neutral, using minimal labeled data. In particular, the study adapts previous work by *Malik et al.* [61] leveraging transformer models in a few-shot setting, as it was shown to achieve significant performance with a large amount of training data. Moreover, the approach can target both functional and non-functional requirements as no assumptions are made about the type of requirements, avoiding the need to filter out non-functional requirements from the dataset. To label each pairing, the proposed method uses pattern-exploiting training (PET) [62] built upon pre-trained language models like BERT trained on a large set of unlabeled pairings and fine-tuned on a small set of labeled ones. The conflict detection dataset used is obtained from *Malik et al.* PET works by rephrasing input requirements, and mapping them to prompts in the form of cloze-style questions. Reformulation is done through masked language modeling, by adding task-specific tokens to the input sequence for conditioning. The rephrasing provides contextual clues to the task being solved, that enhance the model's ability to make accurate predictions with fewer labeled examples. As an example, when determining if two software requirements r1 and r2 are duplicates, the model might be prompted with the question "Does r1 imply r2?" "[MASK]", where the possible predictions for the masked token in this context are 'Yes' or 'No', depending on whether the requirements are duplicates or not. The model computes the probability score associated with each possible value that the token might take, which directly translates into label probabilities.

ID	Pattern
$P_1^*(x)$	“R_1”? $\|$ ______, “R_2”
$P_2^*(x)$	R_1 $\|$ ______, R_2
$P_3(x)$	Given “R_1”, we can conclude that “R_2” is ______.
$P_4(x)$	“R_1” means “R_2”. $\|$ ______.
$P_5(x)$	“R_1” implies “R_2” and “R_2” implies “R_1”. $\|$ The previous sentence is correct: ______.
$P_6(x)$	“R_1” is equivalent to “R_2”. Similarly, “R_2” is equivalent to “R_1”. $\|$ The previous statements are ______.

Table 3.3: PET patterns for conflict detection.

The authors consider different reformulation patterns, investigating both how generic ones transfer into the conflict detection task, and reformulations targeted specifically towards conflict detection. The function that maps the model's predicted output for the masked token to a true label is called verbalizer. Following the previous example, the verbalizer might map "Yes" to the label "Duplicate" and "No" to "Not Duplicate". A PVP is the combination of a pattern and a verbalizer; it defines both how to rephrase the input and how to interpret the model's output, and it's crucial for the performance of PET. By using multiple PVPs and training them separately, the model can generate more accurate predictions, which is especially important in low-data environments. The authors evaluate three PVPs and train three MLMs per PVP and average test performance. Then a PET ensemble of the three PVPs is built and compared to the best supervised model.

The experimental results compared the performance of uncased BERT, uncased BERT-large, RoBERTa, RoBERTa-large, ALBERT-v2, and ALBERT-xxlarge-v2. The best-performing model, RoBERTa-large, is used as both the baseline supervised learner and the underlying MLM for applying PET patterns and verbalizers. The PET ensemble is used to generate soft labels for unlabeled data instances. These are then used along with the labeled training data, to train a sequence classifier using RoBERTa-large. The study found that for all training set sizes, PET is superior or comparable to supervised learning in accuracy, macro F1 score, and weighted F1 score, even with a very limited number of labeled examples. The study also finds that custom cloze-style patterns for conflict detection perform similarly to general patterns, indicating that existing patterns can be reused.

In conclusion, prompt-based few-shot learning through PET with well-chosen patterns and verbalizer emerges as an effective approach for conflict detection in SRS documents, particularly in scenarios with limited labeled data, suggesting that this method can achieve comparable performance to models trained on significantly larger datasets.

Another type of ambiguity is intra-domain ambiguity, which occurs when the same word has multiple meanings based on the context in which it appears within the same application domain or in a multidisciplinary/multi-domain project document. This challenge is addressed by *Moharil and Sharma*[28]. Previous approaches leveraging word embeddings failed to detect this kind of ambiguity as they create static representations, meaning a single context-independent embedding for each word in an input text, thus assuming a word is used in a single context within a document.

The authors present a transformer-based contextualization tool named TABASCO, designed to detect and identify intra-domain ambiguities in software requirements and other project-related documents written in natural language. The tool leverages BERT for generating contextual embeddings for each instance of a noun in the document, thus capturing the context-specific usage of the word. These embeddings are then fed into a K-means clustering algorithm to identify the different contexts (represented by clusters) in which a noun has been used within a single domain, that is, the various instances of a target noun are grouped into different clusters based on their contextual similarity.

The proposed tool pre-processes the input documents through conversion into a suitable format, tokenization of the text, and POS tagging, particularly focusing on nouns. Then BERT generates context-specific embeddings for the target nouns and a similarity matrix is constructed using cosine similarity measures between the vector embeddings of the target noun. K-means clustering is applied to the similarity matrix to identify the distinct contexts in which the noun is used and the most similar words are computed for every occurrence of the target noun in each sentence in each of the predicted clusters.

One of the main strengths of TABASCO is that it does not require training a machine learning model from scratch using huge amounts of project-related labeled data. As a result, it can be directly used by a requirements engineer, business analyst or project manager for identifying potentially ambiguous nouns present in software requirements or other project-related documents.

TABASCO offers several features to enhance flexibility, as, for example, selecting the number of top nouns to be analyzed or the threshold values for creating the size of the similarity matrix and for labeling two different terms as similar. It also allows to generate a summary report and a detailed report with details regarding the identified contexts of meaning for a noun, most similar words for each context and example sentences from the corpus highlighting the context-specific interpretation of the target noun.

TABASCO is validated on a Computer Science specific corpora created by crawling the corresponding category on Wikipedia, and a large-sized multidisciplinary/multi-domain corpora created by merging the software requirements documents available on the PURE dataset. The first case study enables checking whether the tool can correctly identify the multiple contexts in which commonly used nouns have been used in a project document of a single application domain, whereas the second evaluates the efficacy of the tool to differentiate between multiple contexts of commonly used words in a multidisciplinary/multi-domain project document.

The results demonstrate the tool's effectiveness in identifying different contexts for ambiguous nouns discerning differences in meaning both within a single domain and across multiple domains. In summary, TABASCO provides a powerful, user-friendly tool for detecting ambiguities in software requirements documents, offering significant flexibility to users while delivering accurate, context-specific analysis.

Satpute and Agrawal[29] focus on the detection of ambiguities deriving from the use of sarcasm, irony, or contextual nuances. The framework leverages BERT and a Bi-LSTM network to develop a model for estimating whether new, unlabeled statements belong to the sarcastic or non-sarcastic class, where BERT is specifically used for identifying contextual ambiguities while the Bi-LSTM network for detecting sarcasm and irony ambiguities.

The methodology leverages a multi-task learning approach where the two tasks are trained simultaneously, which allows for more efficient training, reduced risk of overfitting and improved learning speed, while also alleviating the computational demand and need for large-scale datasets of deep learning models.

The model uses a training dataset comprising labeled remarks collected from Kaggle. After pre-processing the input sentences and obtaining their word embeddings, these are fed into a shared fully connected layer used for both the sentiment and sarcasm classification tasks, to derive the sentence representation. The output of the shared layer goes into the dense layers of the two models, where a specific activation function is applied for each task (softmax activation

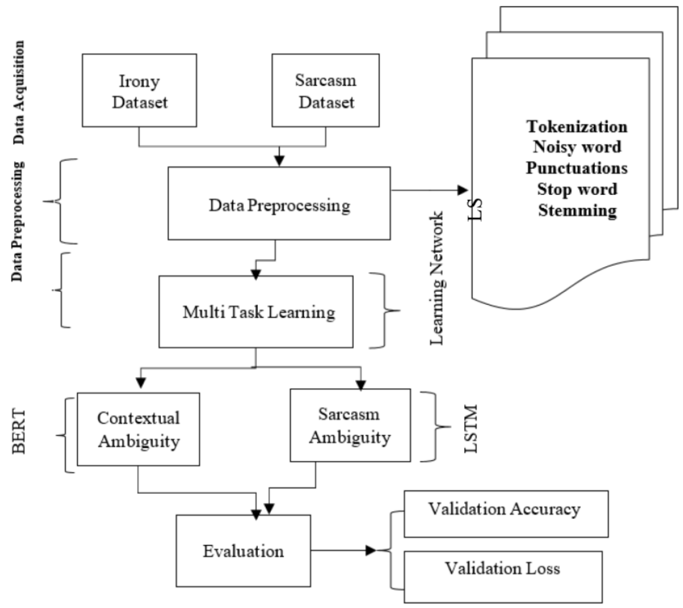


Figure 3.20: Proposed framework.

for sentiment classification and sigmoid activation for sarcasm classification). The framework is designed to allow the secondary task to assist in training the primary task by using a combined loss function that reflects performance on both tasks. This way the model can optimize for both tasks while also improving training efficiency by leveraging additional context from the secondary task.

The results demonstrate that the proposed framework combining BERT and Bi-LSTM in a multi-task learning setup achieves high accuracy, precision and recall, indicating its effectiveness in classifying whether sentences are ambiguous or not, while also rarely missing ambiguous sentences; and a high F1-score, indicating that the model is balanced. The proposed framework also shows superior accuracy when compared with models like SVM and LSTM.

The findings suggest that this approach could be extended to other types of ambiguities and evaluated across diverse datasets, paving the way for more sophisticated NLP systems capable of understanding the subtleties of human communication.

Fantechi et al.[30] explore how recent AI tools like ChatGPT compare to traditional rule-based systems in ambiguity detection tasks. The ChatGPT version used is based on GPT3, whereas QuARS (Quality Analyzer for Requirement Specifications)[63] is taken as a representative tool for the rule-based approach. Rule-based systems rely on deterministic rules to analyze text and identify ambiguities based on lexical and syntactical analysis without taking into consideration the semantic part. On the other hand, ChatGPT, as a large language model, does not use predefined rules but generates responses based on its training data. Although LLMs are known to present issues with explainability, transparency, and consistency in their outputs, the purpose of the study is merely to establish whether the answers provided by the model are reasonable.

The study uses requirements documents including two toy examples created by the authors (a coffee machine and an e-shop) and two third-party examples (a library system and a smart home

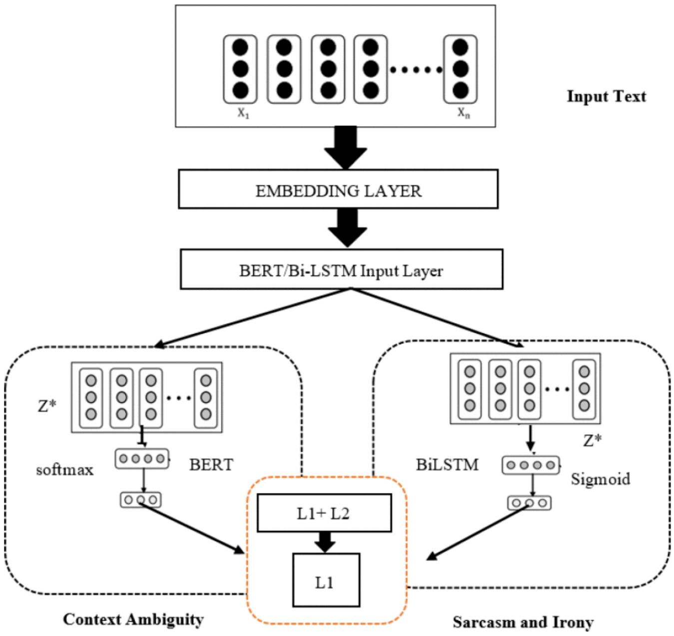


Figure 3.21: Multi-task learning mode.

system).

Each requirement document is input to the models. QuARS provides a structured output indicating the ambiguous requirements and the specific terms causing ambiguity, while ChatGPT's responses are less structured and vary between sessions. The authors then review the outputs from both tools, manually analyze the results, and assess their performance using precision and recall metrics.

The experiments conducted on the four case studies show that both tools detect certain ambiguities, but also miss some and generate false positives, especially when dealing with complex examples. ChatGPT's performance is quite variable, particularly in recall, while more stable and comparable to rule-based approaches in precision. One way to improve recall is running several sessions with the same question. Moreover, ChatGPT struggles with long requirement documents, even if this limitation can be easily overcome by splitting the input document as ambiguity detection does not require to process it as a whole. QuARS shows more consistency, even though in some of the case studies ChatGPT manages to identify hidden ambiguities that QuARS and manual analysis miss, which highlights its potential as a complementary tool.

To conclude, while ChatGPT can be used for ambiguity detection in requirements, its performance is inconsistent and varies between sessions, whereas rule-based tools like QuARS offer more stability but might miss certain types of ambiguities that ChatGPT can detect, due, for example, to relying on incomplete dictionaries and lacking semantic understanding. The authors suggest future research should involve third-party reviewers to mitigate biases, expand the number of documents tested, and explore ChatGPT's ability to detect other defects like incompleteness and inconsistency.

Regarding requirements completeness, the RE literature [31] distinguishes between internal completeness, which refers to the requirements being self-contained, covering all functions and qualities that can be inferred directly from the requirements themselves; and external completeness, which ensures that the requirements include all relevant information drawn from external sources, such as stakeholders or related documents like higher-level requirements and system descriptions. Even though external completeness is not absolute since the external sources might themselves be incomplete or not fully known, when available, they are valuable for identifying missing information in the requirements. Previous approaches for requirements completeness checking typically cross-validate against an external model, use existing development artifacts as a source of knowledge, and focus on a specific domain with a predefined conceptual model, such as privacy policies. In contrast, the approach in this study utilizes a generative language model, particularly BERT in its base-cased version, leveraging its extensive pre-training data to make contextualized predictions. This method does not rely on user-provided artifacts and is not limited to any particular application domain, offering a more flexible and broadly applicable solution to the challenge of completeness checking.

Luitel et al.[31] explore the potential of using LLMs as external source of knowledge to assess requirements completeness and make recommendations on how to make them more complete. In particular, BERT is chosen for its masked language modeling task, which enables it to generate context-aware predictions for missing words, making it well-suited for identifying incomplete or missing information in requirements specifications. Missing information is simulated by randomly withholding parts of a given requirements specification, and the remaining part is used to generate predictions for the masked tokens (the requirements in the disclosed portions are presented to BERT one at a time). The approach aims to balance coverage and noise by optimizing the number of predictions per mask, allowing to generate valuable predictions while reducing the risk of producing irrelevant or excessive suggestions.

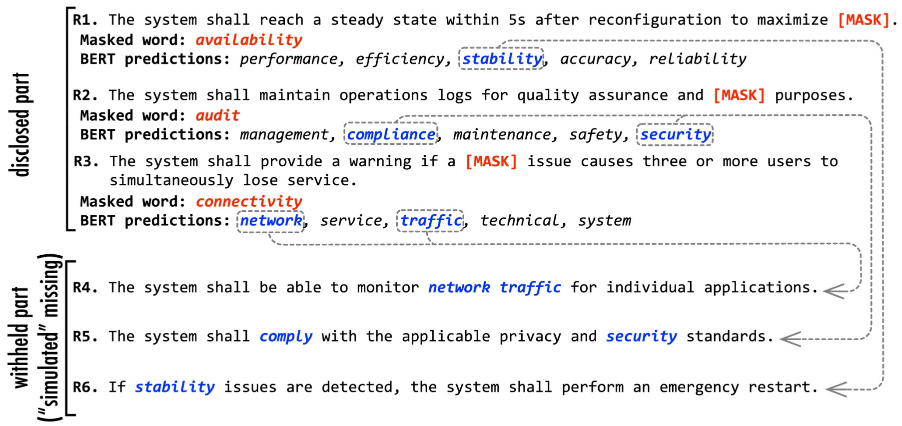


Figure 3.22: Illustrative requirements specification split into a disclosed and a withheld part. The withheld part simulates requirements omissions. Masking words in the disclosed part and having BERT make predictions for the masks reveals some terms that appear only in the withheld part.

The approach involves several steps: The first is parsing the input Requirements Specification using an NLP pipeline to generate annotations such as tokens, sentences, part-of-speech tags, and lemmas. Then, based on the POS tags from the previous step, one noun or one verb at a time in the RS is masked and each generated sentence is fed into BERT, which creates a set of predictions for the masked word, along with a probability score for each prediction, indicating its level of confidence. The third step is filtering out clearly irrelevant predictions, such as predictions of words that are either already present in the disclosed portion of the RS, or common, non-informative terms like "any", "other", or "each". However, to handle the remaining noise in the predictions after these basic filtering operations, a supervised ML-based filter is employed to further refine BERT's predictions and help distinguish relevant from non-relevant predictions. To determine the relevance of each prediction, the withheld portion of the document (which contains the actual missing terms) is compared to BERT's predictions using a cosine similarity measure over word embeddings computed for each prediction. If a predicted word has a similarity of 85% or more to a term in the withheld portion, it is labeled as relevant; otherwise, as non-relevant. These labeled predictions, across all documents in the dataset, are then aggregated into a single training set, a feature matrix where each row represents a prediction and each column represents a feature (including, for example, part-of-speech tags of both the masked word and the predicted term, their lengths, cosine similarity over word embeddings, and TF-IDF scores derived from a domain-specific corpus automatically generated using WikiDoMiner). This training set is used to train an ML-based classifier to distinguish between useful and irrelevant predictions, and, since the features are designed to be generic and normalized, this ensures that the ML model can generalize across different and unseen documents and domains. The final output of the ML-based classifier is the list of BERT predictions that are classified as "relevant".

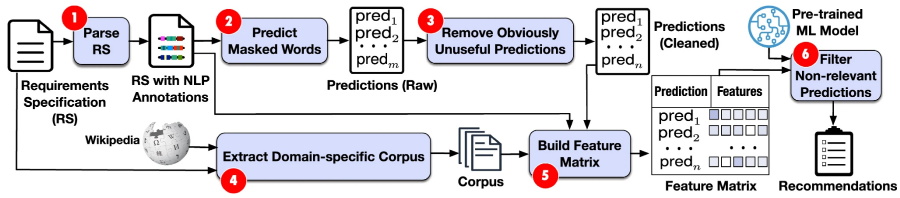


Figure 3.23: Approach overview

The evaluation is performed on a set of 40 requirements documents from the PURE dataset and involves four experiments.

Experiment 1 assesses how accurately BERT can predict missing terminology in a requirements document where 50% of it is withheld. They tested four different numbers of predictions per masked word (5, 10, 15, and 20). The results showed that as the number of predictions increases, coverage (the proportion of missing terms correctly predicted) improves, but accuracy (the percentage of correct predictions) drops. The optimal number of predictions per mask to achieve the best balance between coverage and accuracy is found to be 15.

Experiment 2 compares BERT's performance to simpler baselines using common words, domain-specific terms obtained via TF-IDF-based methods, and synonyms from WordNet, showing that BERT's contextualized predictions outperform all three baselines in terms of both accuracy and coverage.

Experiment 3 tests different ML algorithms for the ML-based filter, while also applying under-sampling and cost-sensitive learning[64] techniques to handle the imbalance between relevant and non-relevant terms, improve recall and reduce the risk of filtering out useful information. Classification is thus performed in different scenarios: using the full training set (strict filtering), under-sampling the training set without CSL (moderate filtering), and under-sampling the training set while also using CSL (lenient filtering). Among the various ML algorithms evaluated, Random Forest and Support Vector Machine show the best accuracy for the task at hand. In particular, for strict and moderate filtering, RF performs best overall for filtering noise, reducing false positives, and ensuring precision; for lenient filtering, RF still achieves higher accuracy and precision, but SVM shows a moderate advantage in prioritizing recall, allowing more relevant predictions to pass through, even at the cost of some noise. This is particularly useful when the goal is to ensure that important relevant terms are not missed, even if it increases the risk of false positives. This combination offers flexibility in the filtering process, enabling the system to adapt to different scenarios by balancing noise reduction with recall optimization.

Finally, in Experiment 4, the full approach is tested on unseen documents to evaluate how well it can generalize. The experiment uses both a 50-50 split scenario (major incompleteness) and a 90-10 split scenario (minor incompleteness). The results showed that BERT was able to predict around 40% of the missing terminology across both split strategies. However, the 50-50 split had much higher accuracy, as there were more missing terms for BERT to predict, while the 90-10 split resulted in lower accuracy due to fewer missing terms. Applying the filtering strategies seen above helped improve accuracy, but at the cost of reducing coverage, so the choice of the filtering strategy depends on what is the best trade-off between minimizing noise and maximizing the discovery of missing terms based on the user's needs.

So in conclusion, while BERT's masked language is shown to potentially improve the completeness of requirements, there are still some limitations in the approach: the use of simulated omissions and the size and diversity of the dataset, which may not be fully representative of real-world scenarios; the reliance on the original BERT pre-trained model without exploring its variants or other LLMs such as the GPT family; the fact that the approach does not yet develop a user-facing tool, nor has it been validated through user studies. Overall, while the findings are promising, they represent an initial step, suggesting the need for further research to fully assess the model's utility in real-world applications.

Systems and software requirements, commonly written in natural language, are prone to issues like ambiguity, incompleteness, and contradictions. Traditional methods, relying on rule-based systems and classical machine learning, face limitations in scalability and semantic understanding. RQ3 examines recent advancements using pre-trained models like BERT, assessing their effectiveness in handling ambiguity and completeness tasks.

Ezzini et al.[25] test multiple solution strategies for anaphoric ambiguity detection and resolution: supervised ML-based approaches trained on either language features from the existing literature on anaphora resolution or feature embeddings extracted from BERT and SBERT; a solution based on existing NLP coreference resolvers; $SpanBERT_{NLP}$, fine-tuned on the CoNLL2011 dataset comprising annotations relevant to anaphoric ambiguity analysis; $SpanBERT_{RE}$, with intermediate fine-tuning on CoNLL2011 and final fine-tuning on a dataset collecting 22 industrial requirements specifications from eight different application domains. The best-performing models for ambiguity detection and anaphora resolution respectively are the ensemble of ML algorithms combining language features and feature embeddings, and $SpanBERT_{RE}$. Intermediate fine-tuning on $SpanBERT_{RE}$ is crucial for performance. Overall, a hybrid solution combining the two would be the optimal choice.

Jafari et al.[26] examine BERT and its variants (RoBERTa, DistilBERT, DeBERTaV3) for anaphoric ambiguity detection, comparing them with the majority classifier and the SVM classifier on top of feature extraction methods such as TF-IDF, GloVe Word Embeddings, and SBERT, as baselines. BERT-based models outperform the baselines, but performance is limited by the small supervised datasets used, suggesting potential improvements with more thorough fine-tuning or techniques like few-shot and contrastive learning. For ambiguity resolution, encoder-based extractive QA models (BERT, RoBERTa, SpanBERT, DistilBERT) outperform encoder-decoder-based text-to-text models (T5-small, T5-base), likely due to issues like hallucination and the use of small variants of the generative models. RoBERTa performs best across all datasets, even surpassing generative decoder-only models GPT-3.5 and GPT-4. While BERT-based models excel, generative models like T5 and GPT show potential but need refinement for requirements engineering.

Helcmezi et al.[27] propose a prompt-based few-shot learning approach for detecting conflicts in software requirement specifications using minimal labeled data. To label each requirements pair as either conflicting, duplicate, or neutral, the proposed method uses pattern-exploiting training (PET) built on pre-trained models like BERT trained on a large set of unlabeled pairs and fine-tuned on a small set of labeled ones. By rephrasing input requirements into cloze-style prompts, PET provides contextual clues that enhance accuracy in predictions with fewer labeled examples. RoBERTa-large is chosen as both the baseline supervised learner and the underlying MLM for applying PET. The PET approach is superior or comparable to supervised learning for all training set sizes, even with a very limited number of labeled examples. The choice of pattern and verbalizer pair has a considerable impact on the performance. Moreover, custom cloze-style patterns perform similarly to general patterns, indicating the reusability of existing patterns.

Moharil and Sharma[28] introduce a transformer-based tool, TABASCO, to address intra-domain ambiguity. The tool leverages BERT to generate contextual embeddings for each instance of a noun in the document, capturing the context-specific usage of the word. These embeddings are then fed into a K-means clustering algorithm to identify the different contexts (clusters) in which a noun has been used based on their contextual similarity. The approach does not require training a model from scratch with huge amounts of project-related labeled data and thus can be directly used, while also offering several features to enhance flexibility, like setting the number of top nouns to be analyzed or the threshold for labeling two different terms as similar. TABASCO is validated on a Computer Science corpus from Wikipedia, and a large-sized multi-domain corpus from the PURE dataset. The results demonstrate the tool's effectiveness in identifying different contexts for ambiguous nouns discerning differences in meaning both within a single domain and across multiple domains.

Satpute and Agrawal[29] focus on the detection of ambiguities deriving from the use of sarcasm, irony, or contextual nuances. The framework leverages BERT for identifying contextual ambiguities and a Bi-LSTM network for detecting sarcasm and irony ambiguities in a multi-task learning approach where the two tasks are trained simultaneously. A shared fully connected layer is used for both the classification tasks, to derive the sentence representation of the input sentences. The output of the shared layer goes into the dense layers of the two models, where a specific activation function is applied for each task and then a combined loss function is computed, that reflects performance on both tasks. The proposed framework achieves superior accuracy when compared with models like SVM and LSTM, with high precision, recall, and F1-score, indicating a balanced model.

Fantechi et al.[30] compare ChatGPT based on GPT3 to traditional rule-based systems exemplified by QuARS[63] at ambiguity detection tasks, to assess whether the answers provided by the LLM are reasonable. Both approaches detect certain ambiguities, but also miss some and generate false positives, especially when dealing with complex examples; QuARS provides a structured output with the ambiguous requirements and the specific terms causing ambiguity, thus offering more stability and consistency. ChatGPT's responses are less structured and variable in performance, particularly in recall, which can be improved by running multiple sessions with the same prompt. ChatGPT sometimes manages to identify hidden ambiguities that QuARS and manual analysis miss, due, for example, to relying on incomplete dictionaries and lacking semantic understanding. This highlights ChatGPT's potential as a complementary tool.

Luitel et al.[31] use BERT for requirements completeness checking without relying on user-provided artifacts or domain-specific restrictions. BERT simulates missing information by withholding parts of requirements and makes predictions for the masked tokens, which are then labeled based on their cosine similarity with the original terms using a threshold of 85%. A supervised ML-based filter trained on the labeled data refines BERT's output, returning only relevant predictions. Testing on 40 PURE dataset documents shows that making 15 predictions per mask balances accuracy and coverage. BERT outperforms baselines using common words, domain-specific terms obtained via TF-IDF-based methods, and synonyms from WordNet, in both metrics. The ML-based filter handles class imbalance using under-sampling and cost-sensitive learning [64]. Three filtering modes are tested —strict (using the full training set), moderate (under-sampling the training set without CSL), and lenient (under-sampling with CSL)—. Random Forest (RF) excels in strict and moderate filtering, reducing false positives and improving precision, while Support Vector Machine (SVM) is better for lenient filtering, thus better preserving recall, allowing more relevant terms through, though with increased noise. This enables flexible balancing between noise reduction and recall. When tested on unseen documents using 50-50 and 90-10 split scenarios, BERT predicted about 40% of missing terms in both cases, with higher accuracy in the 50-50 split. Filtering strategies improved accuracy but reduced coverage, requiring a trade-off based on user preferences between minimizing noise and maximizing term discovery.

In conclusion, despite challenges remain, recent advancements using large language models have brought significant improvements in ambiguity- and completeness-related tasks. These improvements are especially driven by encoder-only pre-trained models like BERT, while generative models like GPT and T5 tend to underperform, and the findings also indicate the effectiveness of exploring techniques such as prompt-based few-shot learning and hybrid approaches combining more traditional methods with PLMs.

3.2.4 RQ4: How can LLMs be leveraged for enhancing requirements traceability?

Requirements traceability is the ability to systematically interrelate each requirement in a project to its origin, dependencies, and related documentation, thereby promoting transparency and accountability throughout the software development process[41] and are critical for various software maintenance tasks like bug localization, bug prediction, and feature location [37].

Automating this process poses significant challenges, such as accurately capturing and maintaining complex relationships among various requirement artifacts, integrating traceability tools into existing workflows, managing inconsistencies in terminology and structure across different domains, and achieving scalability and efficiency, especially in large-scale and complex software projects. Requirements traceability is interpreted as a sentence-pair classification task in natural language, where the semantic similarity of a pair of requirements is measured to determine whether a trace link exists between artifacts, based on the text features of requirements.

Earlier work in automated Trace Link Recovery (TLR) is based on information retrieval (IR) techniques like the Vector Space Model[65], Latent Semantic Indexing[66], and Latent Dirichlet Allocation[67], which strongly rely on feature engineering and manual rules, lack semantic understanding and are hard to generalize. On the other hand, traditional ML-based approaches extract semantic features from artifacts and build classification models such as Naive Bayes and Random Forest to predict the trace link between requirements pairs. Even though these ML-based methods do alleviate to some extent the issues in previous works, they do not eliminate them. More recently introduced deep learning-based methods[68] address both the limitation in semantic understanding and the reliance on manual selection in the feature representation stage, however they still result unsuitable for the traceability link recovery task in many scenarios, mainly because they rely on vast amounts of labeled data, which is scarce and often impractical to retrieve; plus, they generally have a unidirectional structure, which leads to incorporating only one side of the context information; they struggle in effectively embedding context when dealing with longer sequences [35], and are slower than previous IR or ML techniques [32].

Advancements in deep learning technology with pre-trained models like BERT offer promising improvements in the field. Indeed, unsupervised pre-training on a vast amount of unstructured data allows these models to have a deeper understanding of text semantics and to only require a small domain-specific training set for downstream tasks, while also solving the issue of unidirectional context representation.

Lin et al.[32] propose a BERT-based framework called Trace BERT (T-BERT) designed for accurately generating trace links between source code and NL artifacts, including software requirements, even in large-scale scenarios and with limited training data. The authors leverage multi-stage fine-tuning with pre-training, intermediate training and fine-tuning, to transfer knowledge from a closely related Software Engineering project in order to address data sparsity and increase accuracy in the generated links. In the pre-training stage, the BERT model learns the word distribution among NL and PL documents. Specifically, the authors use MS-CodeBert as pre-trained model to handle programming language. In the intermediate-training phase, T-BERT is trained with labeled training examples to perform the code search task, that is, retrieving source code based on an NL description of code functionality, formulated as a binary classification task in which T-BERT identifies whether a given doc-string properly describes its paired function or not. This allows the model to learn general NL-PL classification knowledge, that can be then used to improve the performance on the software traceability task. Finally, in the fine-tuning step, the intermediate-trained T-BERT model is applied to the issue-commit tracing task using real-world OSS datasets.

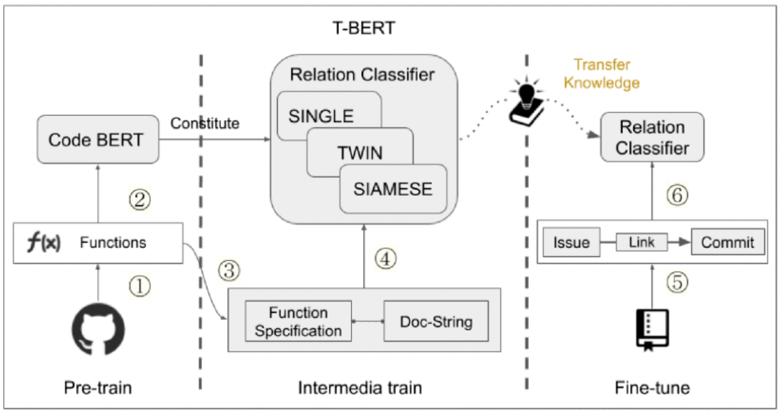
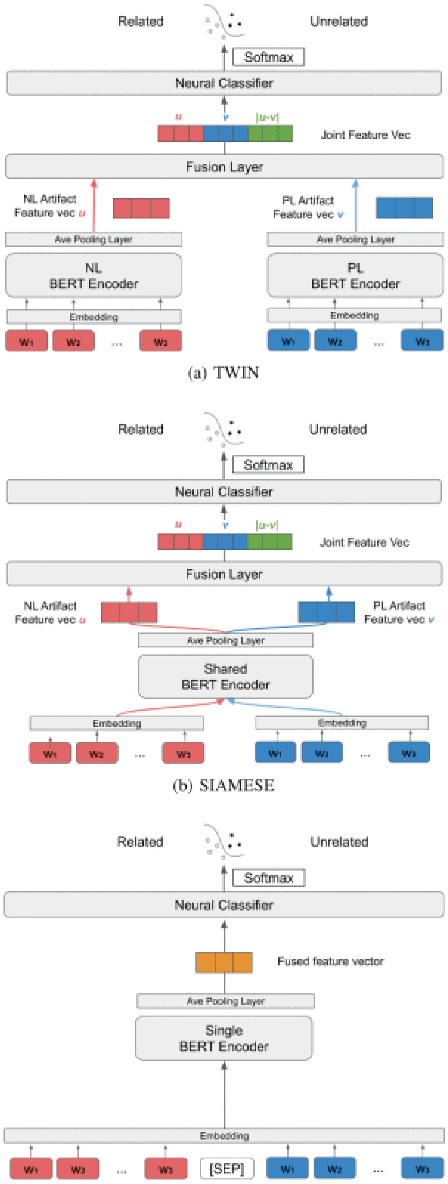


Figure 3.24: A three-step workflow applies T-BERT to NLA-PLA traceability. 1) Pre-training data are functions collected from GitHub projects 2) A BERT is trained as a language model for code with these functions and composed with a relation classifier as the T-BERT model 3) Functions are split as specifications and doc-strings and used as intermediate-training data 4) T-BERT model is intermediate-trained using code search data 5) OSS datasets are collected from GitHub repo 6) T-BERT model is fine-tuned as a trace model using transferred knowledge.

T-BERT is implemented in three architectural variants, with different structures of the BERT classifier: The Single architecture uses one BERT model and a single input sequence where NL and PL texts are concatenated after being annotated with special tokens. The output is a single hidden state matrix that is then passed to the pooling layer to reduce its dimension and to the fusion layer to create a fused feature vector. This feature vector is used by the classification head to predict whether the input NL-PL pair is related or not. The Twin architecture encodes the NL and PL artifacts separately, leveraging two BERT models. The feature vectors representing the artifacts are then computed and concatenated into a joint feature vector for classification. The Siamese architecture uses one BERT model where the artifacts are input sequentially and separate hidden state matrices are created for the two artifact types. Then the joint feature vector is created and sent to classification headers for the prediction task.

All three models with ONS perform well in Code Search and NLA-PLA traceability tasks, with superior performance than baseline IR models like VSM, LSI, LDA, and RNN techniques, and with the Single architecture achieving slightly better results. However, while Single-BERT architecture generates the most accurate links, it is very slow at processing data compared to Siamese and Twin Bert, which accelerate the process by decoupling the feature vector creation steps. Siamese-BERT is the best trade-off between accuracy and efficiency, generating comparable results with significantly less execution time. Furthermore, the results show the importance of the three-step training strategy, where intermediate training on the code search task can significantly improve T-BERT's performance on the traceability problem, suggesting that the knowledge learned from text to structure code (function definition) can be effectively transferred to cases with less structured code formats and limited labeled training data.



(a) TWIN

(b) SIAMESE

(c) SINGLE

Figure 3.25: The architectures of the three T-BERT models proposed and evaluated.

Lin et al.[33] address another issue related to automatically creating and maintaining traceability links, that is performing this task in bilingual contexts. The study proposes a deep learning approach leveraging a bilingual version of BERT, referred to as MultiLingual Trace-BERT or MT-BERT, to generate traceability links in software projects where artifacts are written in both Chinese and English.

First, traditional IR methods are evaluated, both with and without the use of Neural Machine

Translation (NMT) as a preprocessing step to translate all artifacts into English, and the results show that NMT does not always improve traceability, instead, it even introduces errors especially when translating terms with specific technical meanings or when context is lost in translation.

MT-BERT first tokenizes both source and target artifacts, converting words and characters (for languages like Chinese) into tokens. These tokens are then converted into embedding representations by an embedding layer and further processed into context-aware semantic feature vectors by the multilingual BERT encoder. These vectors are then concatenated into a fused feature vector that will be used for binary classification by a three-layer feed-forward neural network predicting the likelihood that a traceability link exists between a given pair of artifacts.

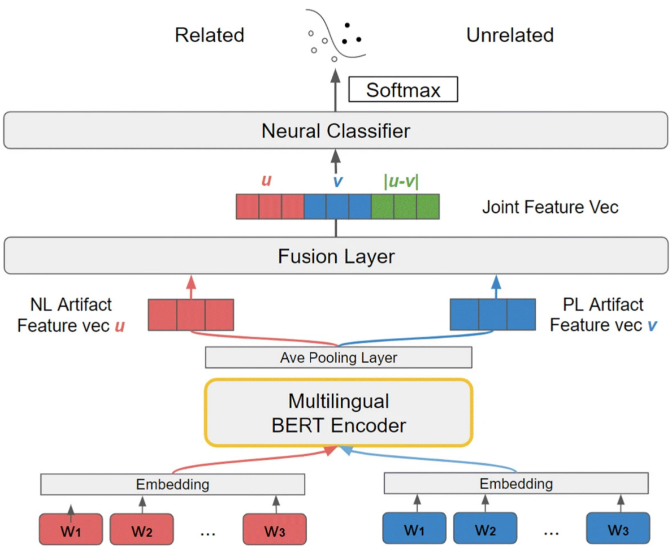


Figure 3.26: Architecture of MT-BERT.

The evaluation compares three different versions of MT-BERT, one using a knowledge-distilled multilingual LM, one with a multilingual LM without knowledge distillation, and one with a monolingual LM. The results demonstrate that MT-BERT leveraging knowledge distillation achieves the best performance among the three, and also with respect to the best performing IR method (generalized VSM with neural machine transition and word embeddings) in larger projects.

Compared to previous works, MT-BERT allows for a deep understanding of NL context, specifically for terms belonging to different languages within the same artifact, and avoids the need for the translation step by processing bilingual artifacts directly. Moreover, the model can leverage pre-existing trace links within a project as training examples to fine-tune the parameters in both the LM and the neural classifier, allowing greater adaptation to a specific project. These results make the approach suitable also in the case of large, complex software projects and industrial environments, representing a major advancement in the field of software traceability in a bilingual context.

Xia and Wang[34] propose a knowledge graph-based method for requirements traceability, where the knowledge graph is used to establish relationships between entries (such as requirements, components, or features) in software documents. Given two input entry sets to be traced, combined with the ontology model, the approach uses a BERT+BiLSTM+CRF[69] model to extract entities and form triplets. BERT is used to generate contextualized word vectors, the Bidirectional LSTM layer outputs the probability of each word belonging to a different label, that is, entity type, and the CRF layer finally incorporates the correlation between predicted labels into the training process, ensuring the validity of the predicted results. The extracted triplets are then used to create the entry knowledge graph, that represents the entities and relationships among them. This allows to accurately find the relevant entries in the next steps by relying on the knowledge graph. Each entry knowledge graph is then merged into a document knowledge graph.

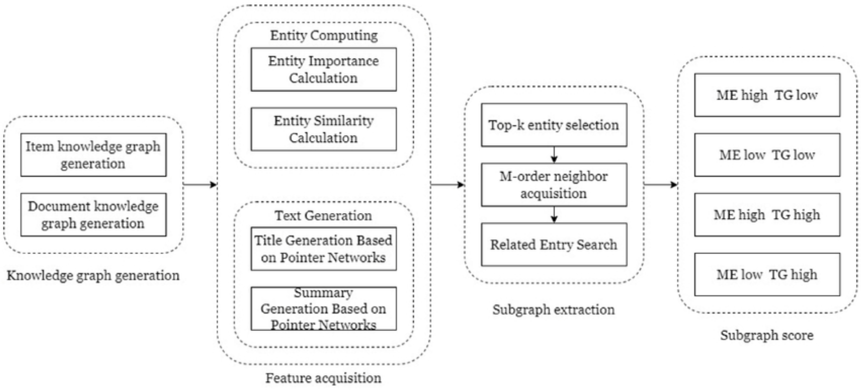


Figure 3.27: Entry traceability algorithm model.

The following steps include feature acquisition, sub-graph extraction, and sub-graph scoring. Feature acquisition involves two parts: calculating the importance of entities and the similarity between all entities in the document knowledge graph using methods from complex network theory, such as the restart random walk and graph kernel; title generation using the pointer generation network to supplement the entries lacking titles to form the knowledge graph, allowing to expand the search scope and discover related entries beyond those that are explicitly identified in the knowledge graph. In particular, on the traditional attention with encoder-decoder framework, at each decoding step, a generation probability is computed, indicating the likelihood of generating a word from the vocabulary rather than directly copying it from the source text, and the final output distribution is produced by combining the vocabulary distribution and the attention distribution through weighting and summation, and predictions are made based on this result. Sub-graph extraction identifies the relevant portions of the knowledge graph (sub-graphs) that relate to a particular entry, based on the relevance obtained by the feature acquisition stage. Finally, sub-graph scoring uses the graph kernel method to compute the similarity between the extracted sub-graphs of related entries and the target entry to determine the traceability relationships.

The experimental evaluations on real requirements documents demonstrate that the proposed method can efficiently and accurately trace requirements from a set of input entries, returning the traceability relationships.

Tian et al.[35] focuses on a less-studied traceability scenario, where the task is to create and synchronously update trace links between high-level requirements (HLRs) and low-level requirements (LLRs), which is particularly challenging in large-scale, rapidly evolving open-source software systems where requirements change frequently. The authors propose a deep-learning-based cross-level requirement tracing model leveraging BERT, called DRAFT ("Deep Requirement

Trace Analyzer Fusing Heterogeneous Features").

The key components of the framework are three and include a project-specific BERT second-phase pre-training module, a heterogeneous feature extraction module, and a trace link identification model that integrates heterogeneous features. These components are executed in sequence. Considering that the requirement tracing task is typically project-specific, DRAFT adopts a two-phase pre-training approach, where BERT is first pre-trained on general data corpora and then fine-tuned on a corpus developed from project-related documents. This allows BERT to better adapt to and understand the context of the requirements within a specific project and ensures that text encoding is highly suitable for the project context. The project-specific pre-training model obtained through second-phase pre-training is then used to extract requirements features.

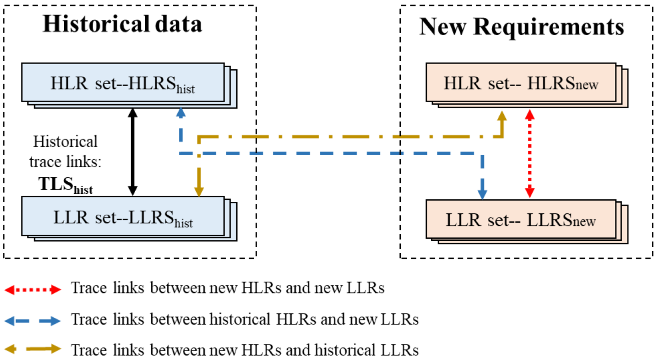


Figure 3.28: Identifying the source and target of the trace link for new requirements.

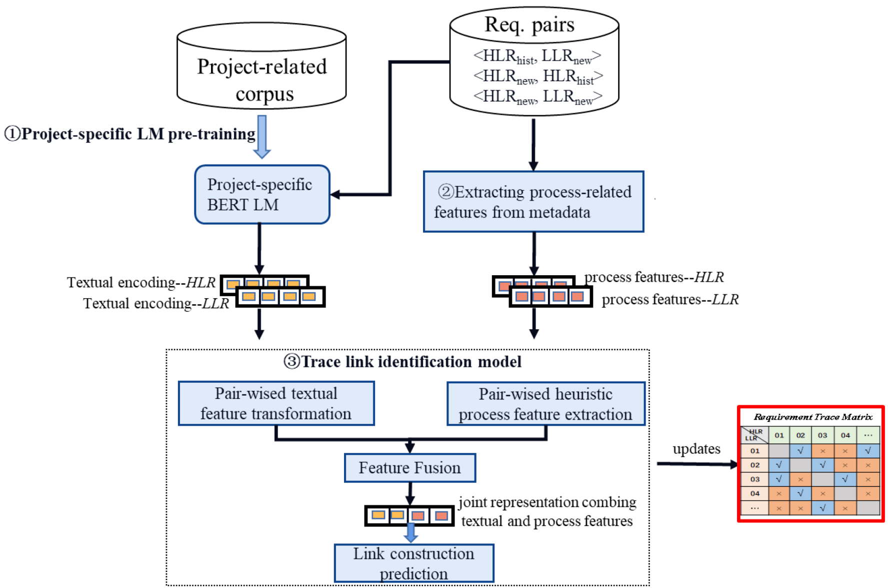


Figure 3.29: Flowchart of DRAFT.

DRAFT considers both the semantic content of requirement descriptions and the process features based on metadata (such as the creator, assignee, and components involved), thus deriving a joint representation of heterogeneous features. For process features extraction, DRAFT integrates 11 heuristic features based on requirement metadata and, besides text and process features, it also uses historical trace link data. In particular, the historical trace list of each requirement is retrieved from the historical trace link and the extended features between the requirements in the historical trace list and the newly added requirements are analyzed.

To allow the joint analysis of these heterogeneous features, the model leverages a neural network model with a requirement-pair feature embedding layer used to embed text and process features; a heterogeneous feature fusion layer used to fuse text and heuristic features to analyze the commonality of a requirements pair in terms of text semantics and process data; a trace link identification layer with two fully connected layers and a softmax output layer that returns whether a trace link is identified between a pair of requirements, based on the joint feature representation of the pair of requirements in the text and process information.

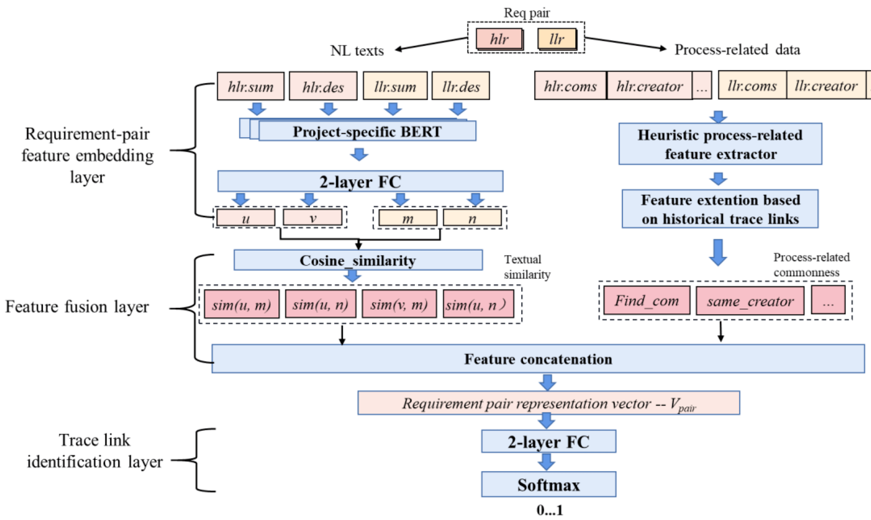


Figure 3.30: The three modules of DRAFT's neural network structure: requirement-pair feature embedding, feature fusion, and trace link identification layers.

The authors collected cross-level requirements and corresponding trace links from real-world open-source projects, developed datasets based on the trace link update scenario for new requirements in practical scenarios, and conducted an experimental evaluation to validate DRAFT's effectiveness against traditional information retrieval algorithms VSM and LSI, relevance feedback technique that represents an improvement of VSM using historical trace links), and TraceBERT. The results demonstrate that DRAFT outperforms the four baseline methods, achieving higher F1 and F2 scores, and confirm the positive impact of including heuristic features based on metadata -as they contain important commonalities between cross-level requirements-, and of applying second-phase pre-training, even though its effect is influenced by the project's requirements set size and the quality of requirements descriptions.

Majidzadeh et al.[36] address another aspect that requirements traceability prediction techniques have to face: the variability in requirements' and traceability links' types and structure, which many approaches do not take into consideration, limiting their generalizability.

The authors propose an approach for multi-type requirements traceability based on pre-trained language models leveraging code augmentation to increase the dataset size and diversity and multi-stage fine-tuning to tailor the model for traceability tasks, while also preparing the model for predicting three types of links including issue-commit, issue-method, and documentation-method links.

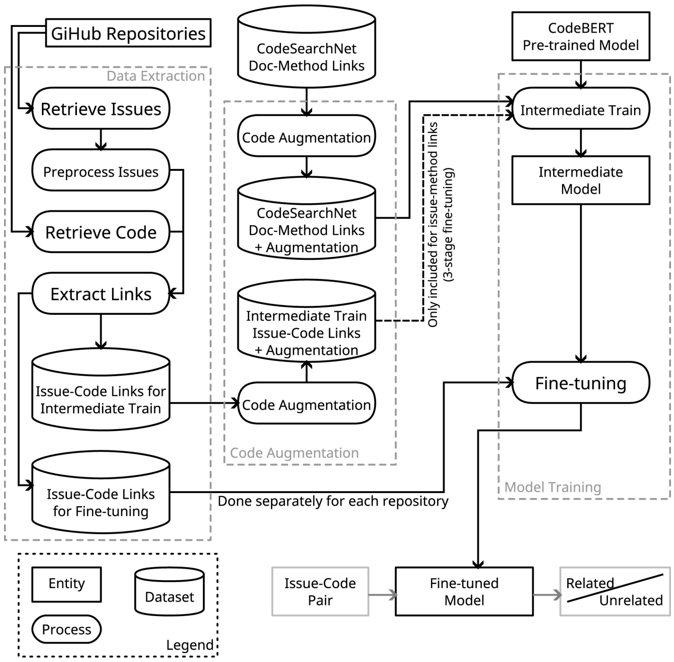


Figure 3.31: Overview of the approach.

The first step in the process involves extracting issue-commit and issue-method links from GitHub repositories, and documentation-method links from the CodeSearchNet dataset to obtain the datasets for training and evaluation of the model. Then, three code augmentation techniques ('rename variable', 'swap operands', and 'swap statements') are applied to increase the variability of requirement-code pairs in each dataset. By combining these transformations, new labeled code snippets to be used in the training step are obtained. The final step is model training. The study leverages MS-CodeBERT and fine-tunes it with 2-stage and 3-stage training. 2-stage fine-tuning involves training the pre-trained model once with the CodeSearchNet dataset and then fine-tuning it separately for each software repository. 3-stage fine-tuning involves a first intermediate training step using the CodeSearchNet dataset, and a second intermediate training step using a portion of the extracted issue-code dataset, before the final fine-tuning. The 3-stage fine-tuning is used to retrieve issue-method links as they are harder to recover because of their indirect relations, and this allows the model to better understand them.

The authors evaluate the effect of code augmentation in recovering the three types of links, and assess the effect of 3-stage fine-tuning in retrieving issue-method links by comparison with T-BERT, which predicts issue-commit links using BERT without code augmentation, and CODFREL[70], which recovers fine-grained links between requirements and code using LSI and an evolutionary algorithm; both adapted to suit the different types of data.

The results reveal that in all cases, the language model-based methods perform significantly better than CODFREL. Code augmentation significantly improves the model's performance in recovering doc-method and issue-commit links, with improvements in precision, recall, and F1

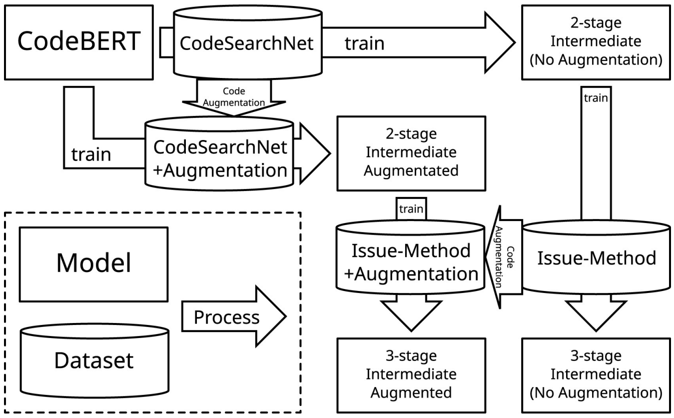


Figure 3.32: The training paths for the intermediate models. The issue-method dataset used here contains a separate set of repositories from the datasets used for the final fine-tunings and evaluations. The final fine-tunings are done separately for each repository.

score over baselines, while, on the other hand, it does not exhibit a significant impact on the performance of recovering issue-method links. This may be due to the larger dataset of extracted issue-method links and/or the difference between issue-method links compared to doc-method or issue-commit links, as issue-method links have indirect relations. 3-stage fine-tuning instead significantly improves the model's performance in recovering issue-method links, the reason being that an additional training stage helps to familiarize the model with such links, that are indirect and extracted from issue-commit links. The lack of improvements with code augmentation combined with the 3-stage fine-tuning may be a result of the longer training process and the larger training dataset. The results thus show that code augmentation is helpful in improving the model's generalization and accuracy when there is no fine-tuning data available, which is often the case in real-world new projects.

The proposed approach is not without limitations: the datasets used lack semantic information about the code outside of the functions (i.e., external dependencies to other parts of the program or libraries) and the applied transformation techniques rely on the syntactic analysis within the scope of the method, which may not preserve the program behavior in all cases. Another aspect is that the model is tested on the Java language, which means that using code segments in other programming languages may impact the performance. Moreover, the actual impact of each code augmentation technique is unknown, as measuring the impact of each technique individually would require much more resources, while the results would most likely be close and not significantly different.

Lan et al.[37] propose BTLink, which focuses on issue-commit traceability link recovery using BERT. Establishing traceability links between issues and commits is particularly important for several software development and maintenance activities, such as bug localization, bug prediction or commit analysis. However, just like other software traceability links, they typically rely on manual work and this leads to incomplete, inaccurate or missing links.

The goal of BTLink is to improve the identification and accuracy of links and effectiveness of the process, addressing the main drawback of previous deep learning approaches in TLR, which is the need for a large amount of labeled training data. BTLink processes both natural language

(NL) and programming language (PL) data from issues and commits, combining them to create more accurate feature vectors that can be used to classify and recover the missing links. The framework includes a BERT embedding layer, a fusion layer, and a classification layer.

BTLink uses two BERT models: RoBERTa for NL-NL pairs (combining issue text and commit text) and CodeBERT for NL-PL pairs (combining issue text and commit code). The two BERT encoders generate feature vectors representing the relationship between issues and commits, which are then concatenated into a joint feature vector by the fusion layer and fed into the classification layer to determine if a link exists between an issue and a commit.

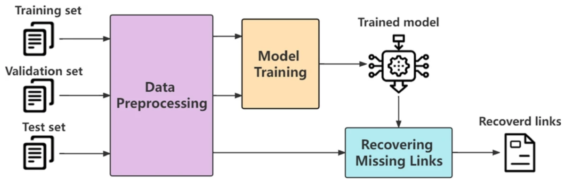


Figure 3.33: BTLink framework.

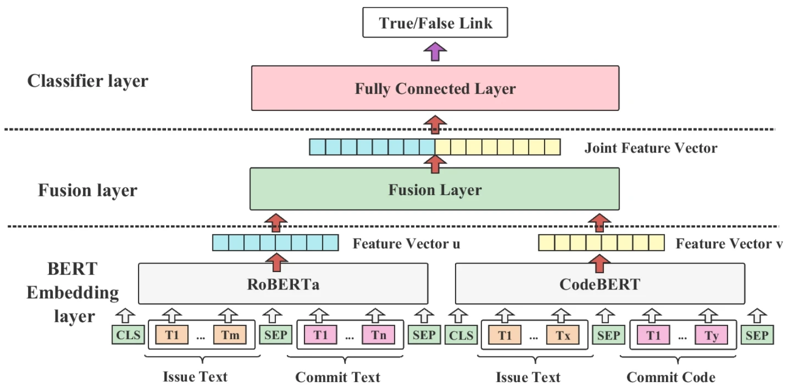


Figure 3.34: Overall structure of BTLink.

The authors conduct an extensive experimental evaluation on 12 datasets from open-source software projects, to investigate various aspects of the proposed approach.

First, two BTLink variants are tested, with different embedding layers and input structure. in the Siamese variant, RoBERTa and CoeBERT are used separately to derive feature vectors from issue text on the one hand, and feature vectors from commit text and commit code on the other hand. In the Shared variant, a single shared BERT model (RoBERTa or CodeBERT) is used for embedding issue text-commit text pairs, and issue text-commit code pairs.

Evaluation results reveal that combining the issue text with the commit text and the commit code respectively as input to the two BERT encoders can best understand the relationship between issue and commit, achieving much better performance compared to Siamese and Shared variants. Another result is that the combined use of NL-NL and NL-PL pairs leads to better results and that both the NL-NL encoder and the NL-PL encoder contribute to the model, although the former has a larger contribution. At the same time, the effect of using the pre-trained versions of RoBERTa and CodeBERT for the two encoders is explored, by assessing the performance of BTLink variants with non-pre-trained models in the embedding layer. Using the pre-trained models can lead to a better representation of semantic relations, thereby contributing to the

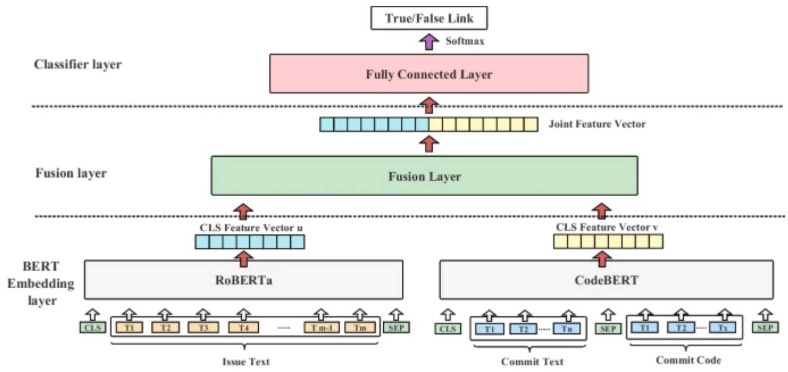


Figure 3.35: Overall structure of SiameseBTLink.

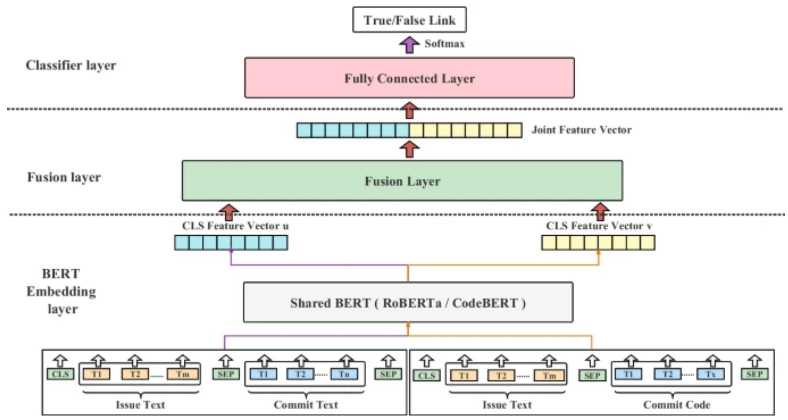


Figure 3.36: Overall structure of SharedBTLink

overall performance of the approach. The authors also investigated substituting the NL-PL encoder with a pre-trained model other than CodeBERT but found that CodeBERT is the most suitable for the proposed way of constructing code information.

Concerning the task of recovering missing issue-commit links, BTLink significantly outperforms state-of-the-art methods like FRLink[71], DeepLink[72] and hybrid-linker[73] considering the overall metrics of precision, recall, F1 score, and others. This reveals that text data in software artifacts is critical for link recovery, and effectively utilizing it is key to success. In particular, semantic associations in text data obtained with BERT's contextualized embeddings prove more helpful for link recovery than text similarity, and BERT conforms its superiority to traditional methods like TF-IDF and Word2vec. In addition, the evaluation reveals that using traditional ML methods —such as SVM, NB, DT, RF, and LR— as the classifier layer in BTLink does not achieve satisfactory results compared to the fully connected layer used, as they struggle to understand the deep features extracted by deep learning techniques. Moreover, BTLink also proves effective in cross-project scenarios, demonstrating its robustness and applicability across different project domains and scales thanks to the transfer learning abilities of pre-trained models.

Zhang et al.[74] again focus on issue-commit link recovery, aiming at addressing challenges still present in works like T-BERT leveraging pre-trained models. These include long training and inference time due to a large number of parameters; neglecting correlations among commits in a one-to-many issue-commit link; the generation of conflicting false links during training; inappropriate modeling of loosely related and unrelated code changes, ignoring that code in a commit can present changes related to different purposes.

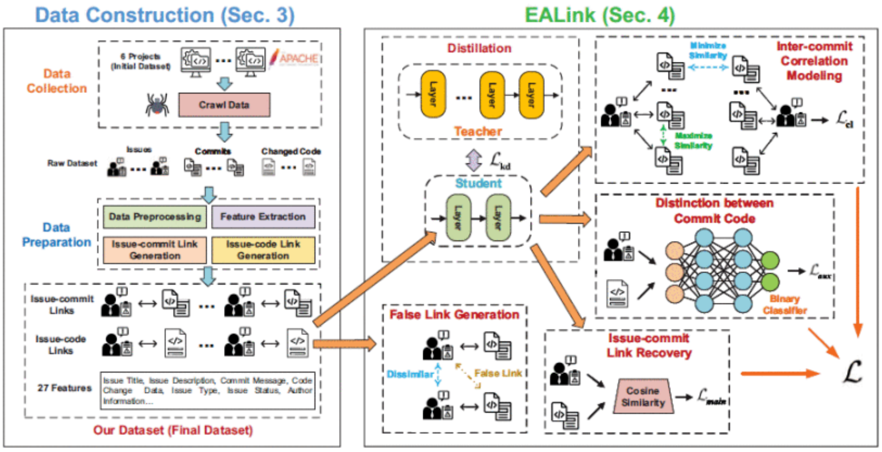


Figure 3.37: Overview of EALink.

Their proposed approach EALink leverages knowledge distillation to build a lightweight model with a reduced number of parameters, shorter training and inference time, starting from a pre-trained teacher model. The resulting model achieves similar performance to the teacher model, with similar representations for NL and PL tokens, but allows to alleviate computational overhead. The default teacher model in EALink is CodeBERT, while the student model is a two-layer RoBERTa.

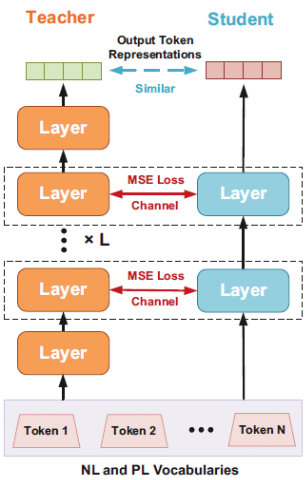


Figure 3.38: Distillation for the code pre-trained model.

After building the student model, EALink fine-tunes it for the issue-commit link recovery task: the model is trained to capture inter-commit correlation via contrastive learning[75], while also being trained on the auxiliary task of issue-code link prediction to distinguish the relevance of different commit code. The two related tasks are trained jointly, in a multi-task learning way.

Finally, to address the generation of conflicting false links, the authors propose a new method for constructing more accurate false links to be used for training, based on cosine similarity rather than using time intervals to filter commits as traditional false link generation mechanisms.

EALink is evaluated against three baseline models -T-BERT, DeepLink and VSM- through extensive experiments on a new large-scale dataset based on 6 Java projects, constructed for the issue-commit link recovery task and the issue-code link recovery auxiliary task. Even though baselines outperform EALink in a few cases, they do not show robust performance. The proposed approach demonstrates higher efficiency and accuracy in issue-commit link recovery than state-of-the-art models, and more suitability on large projects, thanks to its shorter training and test time. Ablation studies reveal that the auxiliary task of issue-code link recovery and the inter-commit correlation modeling component both enhance EALink, improving the quality of generated links, with a larger contribution of the inter-commit correlation modeling component). Additionally, different teacher models are tested beside CodeBERT, including GraphCodeBert and UniXcoder, showing only a slight decrease in performance, which indicates that EALink allows flexibility in the choice of the code pre-trained model used.

Although the encouraging results, EALink has its limitations: while applying knowledge distillation reduces the parameters' quantity and training time compared to models like T-BERT and BTLink, it results in a loss of information, especially when using a single language model for pre-training.

Therefore, unlike traditional knowledge distillation where a single-teacher model is used, MTLink by *Deng et al.*[38] leverages multi-teacher knowledge distillation (MTKD). This allows to build a compressed student model with a smaller number of parameters while not incurring in a significant loss of information, thanks to combining the knowledge learned by models focused on different aspects of data. The two teacher models used are CodeBERT and GraphCodeBERT, with CodeBERT focusing on deep contextual understanding of NL and PL, while GraphCodeBERT extracts code-related information from its syntax tree representation.

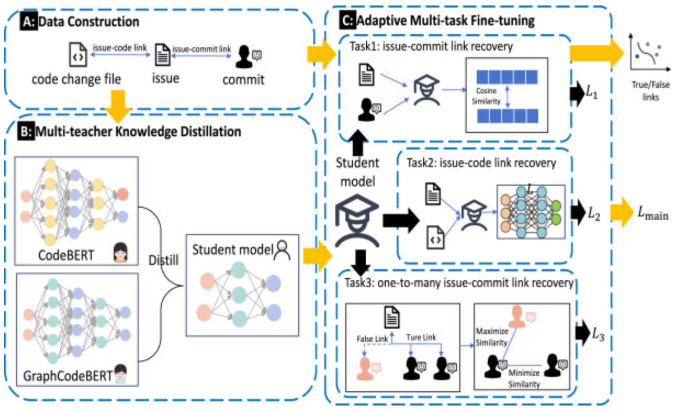


Figure 3.39: MTLink overview.

The student model is trained to mimic the output of the teacher models by minimizing the difference between their internal hidden states. The paper presents two frameworks for knowledge distillation. In the first, Ensemble Teacher 1, the process of transferring knowledge is done by directly comparing the hidden states of each teacher model and the student model at every layer: for each layer, the Mean Squared Error (MSE) between the hidden state outputs of each teacher model and the student model is calculated. The student model learns to approximate each teacher's knowledge by minimizing the error between its hidden layer outputs and the outputs of each teacher model.

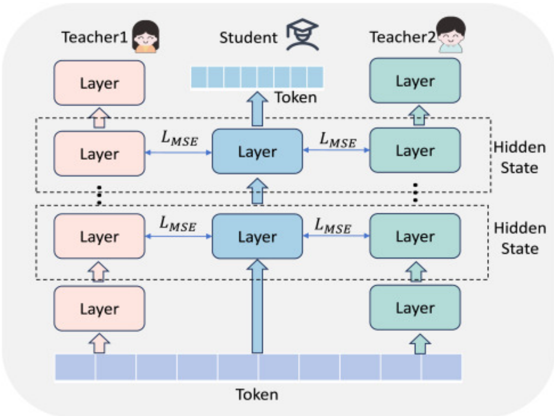


Figure 3.40: Ensemble Teacher 1: Teacher framework for average MSE.

The second framework, Ensemble Teacher 2, takes a slightly different approach by providing the student model with a unified, averaged view of the knowledge from both teacher models, to have it learn a balanced representation from both of them, rather than two separate sets of representations. The method calculates the weighted average of hidden states from all the teacher models and compares the student model's hidden states to this averaged hidden state. The parameters of the student model are optimized so that it can indirectly mimic the teacher models by minimizing the MSE between the student model's output and the averaged hidden state representation from the teacher models.

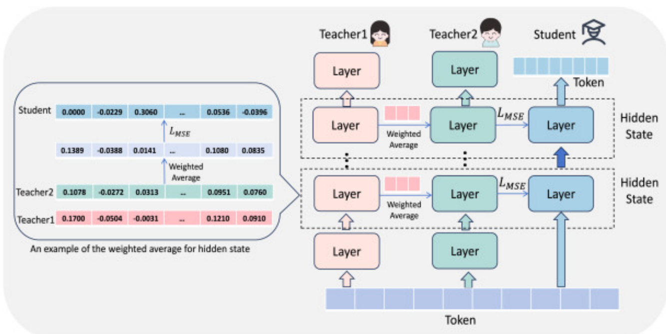


Figure 3.41: Ensemble Teacher 2: Teacher framework for average hidden layer states.

The student model is then fine-tuned through multi-task learning by simultaneously training on different tasks using the same dataset, which enables the model to learn multiple related tasks at the same time, improving model generalization. The first task is issue-commit link recovery, which involves computing the similarity between the two artifacts and minimizing the error between the predicted similarity and the true label of whether the issue and commit are linked or not. The second task is issue-code (code change file) link recovery. This type of link is established based on existing issue-commit links based on the assumption that if an issue contains the name of a code change file, there exists a link between them. This task focuses on enhancing the semantic information of commits by linking issues to the code changes made in the commit, and it is optimized by minimizing the cross-entropy loss between the predicted links and the actual links. Finally, the third task is one-to-many issue-commit link recovery. Since a single issue can be associated with multiple commits, this task helps to improve the model's ability to recognize true links between issues and commits using contrastive learning. For each issue, some of its related commits are randomly chosen as positive examples to represent correct links, while some non-related commits are selected as negative examples, and the model then compares the positive and negative examples, learning to distinguish which commits are truly related to an issue and alleviating the confusion caused by similar artifact. Training the model on these related tasks helps it to better capture the correlations among different artifacts to avoid under-fitting and distinguishing connections between unrelated artifacts to avoid overfitting.

The dataset used for training the model on the issue-commit and issue-code link recovery tasks consists of NL and PL features (like issue and commit ID, issue description, commit message, changed files,...).

To optimize all tasks simultaneously, MTLink weights and combines the above loss functions into an overall loss function, using Dynamic Weight Average (DWA)[76] to adjust the weight of each task dynamically during training. After weights initialization, the loss function of each task is computed and DWA adapts their weights in the combined loss by assigning higher weights to tasks that are learning more slowly (i.e. tasks for which the rate of change of the loss function in the two most recent epochs is lower, meaning loss is still high and decreasing slowly). The combined loss is then calculated with the new set of updated weights and the model parameters are optimized by minimizing this loss function through the gradient descent algorithm. The above steps are iterated at each epoch, continuously adapting the weights and optimizing the model. This ensures that the model performs well across different types of relationships, and prevents it from being dominated by easier activities, while also avoiding the need for manual parameter tuning.

The authors conducted experiments on four open-source projects to evaluate MTLink's performance against four baselines, including EALink, VSM (Vector Space Model), T-BERT and DeepLink, that cover representative tracing methods in both IR and deep learning. The results demonstrate that MTLink outperforms existing state-of-the-art methods thanks to leveraging multi-teacher knowledge distillation (and, more specifically, Ensemble Teacher 1 framework) and adaptive multi-task fine-tuning for balancing the tasks. The approach is particularly suitable for large-scale datasets. The paper also discusses the challenges of overfitting in smaller datasets and the importance of clean data for optimal model performance.

In conclusion, MTLink represents a significant advancement in ILR by combining model compression techniques with adaptive multi-task learning, offering a scalable and accurate solution for software traceability in large-scale projects.

Zhu et al.[39] propose DSSLink to enhance traceability link recovery (TLR) between issues and commits when labeled data is scarce by leveraging deep semi-supervised learning, which combines a small amount of labeled data and a large amount of unlabeled data to infer pseudo-labels for unlabeled pairs.

The proposed model builds upon T-BERT and BTLink, and starts by training them using a limited set of labeled software artifact pairs. This allows the model to predict labels and provide feature representation for unlabeled data, which will serve as inputs for semi-supervised learning. Next, unlabeled software artifacts are paired based on a temporal rule that pairs each issue with all commits that are submitted within its creation and closing time window (thus re-opened issues would be excluded from the pairing process).

After pairing, DSSLink uses semi-supervised learning algorithms (specifically, self-training[77] and label propagation[78]) to estimate whether traceability links between the unlabeled issues and commits exist, by leveraging local features extracted from labeled data and the sample distribution of unlabeled data. The underlying assumptions of semi-supervised learning are that close data points in a high-density region tend to share the same label, data points tend to form discrete clusters and samples of the same cluster are likely to have the same class label, and, finally, two samples close to each other in a high-dimensional space are also likely to be close in low-dimensional manifolds.

Self-training involves using the classifier initially trained on labeled data to assign pseudo-labels to unlabeled data. Then, the pseudo-labeled data points which the model is most confident about are selected based on a confidence threshold to be combined with the original labeled data to retrain the classifier. This iterative process improves the model's ability to generalize from a small labeled dataset, learning general rules from labeled data in the current dataset to predict new data outside the current dataset.

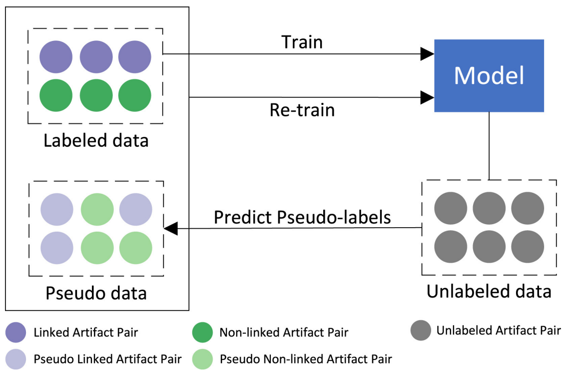


Figure 3.42: Self-training process.

Label propagation begins by obtaining feature representations of both labeled and unlabeled data using the pre-trained models, which are then used to create a k-adjacency graph based on similarity. Labels are diffused from labeled data across the graph to the unlabeled data, thus spreading label information through the network. This process assigns pseudo-labels to unlabeled data points, reflecting the likelihood of a traceability link between the corresponding issues and commits. To account for the uncertainty in these pseudo-labels, weights are assigned based on the entropy of the predicted probabilities. This ensures that samples with higher uncertainty have less influence on the model's training, helping to reduce the impact of incorrect pseudo-labels and improving the overall reliability of the traceability link recovery process.

The vast majority of pseudo-labels inferred for the unlabeled data through semi-supervised learning are negative, which leads to class imbalance and consequently, hinders the reliability of the TLR model. To address this issue, Dynamic Random Negative Sampling (DRNS) by [32] is used to select instances with a pseudo-label of 1 (i.e. unlabeled artifact pairs inferred as positive samples through deep semi-supervised learning) and generate negative samples that are equal in number to the positive samples, thus avoiding the model to be biased towards predicting the majority class (i.e., no link) and ensuring that the pseudo-labeled dataset remains balanced. The

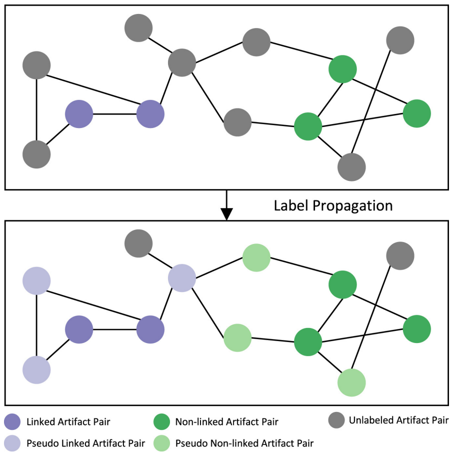


Figure 3.43: Label propagation process.

model undergoes iterative training where, at the start of each epoch, pseudo-labels are inferred for unlabeled data using the pre-trained model. Then labeled and pseudo-labeled data are combined, separate losses are first computed for each, and, finally, a combined loss is used to update the model's parameters.

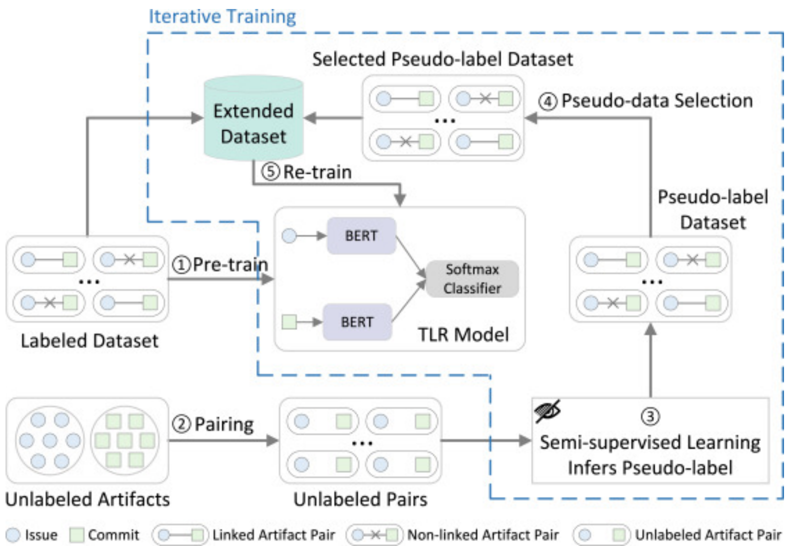


Figure 3.44: DSSLink overview.

DSSLink is evaluated on 15 open-source GitHub and Apache projects against T-BERT, BTLink, and TraceFUN[79], which operates under the assumption that high similarity between an unlabeled and a labeled artifact implies a high probability that the unlabeled artifact is also connected to the linked objects associated with the labeled artifact, but it relies heavily on the availability of labeled data. The results show that DSSLink significantly outperforms both baselines and TraceFUN, particularly when a larger proportion of unlabeled data is used, compared to labeled data. Besides that, the comparison between self-training and label propagation reveals that they deliver similar performance, while the pairing method used for unlabeled data reduces false positives, improving overall model learning and maintaining high performance. Overall, while the method increases computational demands due to the iterative training process, it remains feasible and justifiable given the substantial improvements in TLR performance.

Rodriguez et al.[40] explore the process of prompt engineering to extract link predictions from an LLM. The aim is to assist researchers and engineers in constructing traceability prompts to effectively harness LLMs for advancing automatic traceability.

The optimal prompting strategy varies based on aspects such as the specific model being used -including different versions of the same base model-, available resources and usage scenario, leading to high variability across projects, prompts, and parameters. The objective of the study is to identify common prompt patterns that improve effectiveness within certain projects and trace objectives. The BERT model already brought dramatic improvements to the field of automatic traceability, however, it still shows struggles in scenarios with highly technical domain-specific terminology and limited training data, requiring extensive pre-training on domain-specific documentation to interpret the vocabulary within a project, and thus leading to low accuracy in the trace links in many projects and domains.

The paper aims to improve results in the field by leveraging LLMs, which represent a further advancement in the application of the Transformer architecture, and, more specifically, the Claude model. The experimental setup involves two primary approaches for predicting trace links: A classification task where the model is prompted to classify pairs of source and target artifacts as either linked or not linked. The prompt follows a standard format, asking the model whether a source artifact is related to a target artifact with a "yes" or "no" response; a ranking task where the model is prompted to rank all potential target artifacts for a given source artifact based on their relevance. The ranked list is used to calculate the Mean Average Precision (MAP) score, which measures the effectiveness of the ranking.

The process involves subsequent conversational interactions with the LLM throughout the process, to further refine the prompts. As an example, the model initially generated many false negatives, misclassifying related artifacts that displayed different levels of abstraction, or because it struggled to recognize that they belonged to the same system. Thus the authors modified the prompt, instructing the LLM to ignore abstraction levels and specifying if the artifacts come from the same system. As a result, the model did not generate false negatives but ended up identifying many false positives, probably due to misalignment on what should be considered a trace-link, rather than to misunderstanding. Thus, the prompts were further refined to better grasp the relationships between the artifacts by focusing on the specific relationships between requirements, asking if one artifact decomposes or fulfills another, to help capture hierarchical and functional dependencies. Further improvements include leveraging a chain-of-thought framework by instructing the LLM to break the task down into reasoning steps to help the model arrive at a more informed conclusion. The authors modify the prompt to ask the model to first give a reason why the artifacts might be related and a reason why they might not be related. Based on this, the model provides a final "yes" or "no" answer regarding the existence of a trace link. This method improved both the precision and recall of the model's predictions by adding a layer of reasoning that allowed the model to think through the decision more carefully. Finally, a final prompt was created, combining each of the questions asked in previous prompts and chain-of-thought reasoning to determine whether the artifacts were related. In the ranking prompt approach, the model is tasked with ranking all potential target artifacts for a given source artifact in order of relevance. Initially, the authors found that when artifacts were presented in a random order, the model's performance was barely above random chance, whereas, by pre-sorting the artifacts

Human: I am giving you two software artifacts from a system. Your job is to determine if there is a traceability link. Answer whether (2) implements a part of (1) with yes or no enclosed in <implements> </implements>. Answer whether (2) is a hierarchical decomposition of (1) with yes or no enclosed in <decomposed> </decomposed>. Answer whether (2) fulfills (1) with yes or no enclosed in <fulfills> </fulfills>. Answer whether (2) and (1) are part of the same feature and shares functionality with yes or no enclosed in <feature> </feature>. Answer whether (2) and (1) are dependent on the same system capability with yes or no enclosed in <capability> </capability>. Use your answers to give one reason why (1) might be related to (2) enclosed in <related> </related> and one reason why (1) might be un-related to (2) enclosed in <unrelated> </unrelated> Now answer is (1) related to (2) with yes or no enclosed in <traced> </traced>.

Figure 3.45: Final prompt combining all questions and chain-of-thought reasoning.

based on their similarity to the source using a traditional vector space model (VSM), the LLM was able to build on this initial ranking and improve performance. The ranking approach allows for a more nuanced understanding of how closely linked the artifacts are, compared to a binary decision, expressing the degree of confidence of the prediction.

The study shows how even slight changes in the prompts can importantly affect the output, thereby highlighting the role of a careful prompt design. For example, techniques like chain-of-thought reasoning and the specification of the target use case of the traceability links help in generating more consistent and coherent results. Still, the performance of a given prompt with respect to alternative phrasings varies across datasets and models.

Despite these limitations, the results obtained are encouraging: the model achieves high MAP scores without performing any additional pre-training and also provides in-depth explanations of its decisions, which are useful in the iterative refinement of prompts. The study provides promising evidence that prompt engineering can enhance LLMs for software traceability tasks.

Requirements traceability is also crucial when it comes to aligning text-based requirements with goal models during the requirements engineering phase. Establishing traceability links between requirements and higher-level goals enables tracking their evolution and analyzing the impact of changes.

Forward traceability connects goals to the requirements derived from them, ensuring that each goal is addressed by one or more requirements, whereas backward traceability links requirements back to the goals they support. Cross-cutting traceability, instead, establishes connections between goals and requirements that may not follow a direct hierarchical relationship but are related, such as linking security goals with requirements for encryption and access control. *Hassine*[41] focuses on using LLMs and prompt engineering to generate security-oriented traceability links between requirements expressed in natural language and goals represented in textual Goal-oriented Requirement Language (TGRL)[80] model.

The LLM in use is GPT-3.5-turbo, and requirements and goals, extracted from a text file and a .xgrl file respectively, are its inputs. The prompt design includes the task description -determining links between security-related requirements and security-related goals, and providing an explanation for the result-; it informs the LLM of the possibility of one-to-many relationships between goals and requirements and vice versa; and it sets a constraint to only focus on security-related items. Also, the approach uses a zero-shot strategy, role prompting, and a temperature value of 0.2. The authors experiment with prompt variations in order to enhance response accuracy and achieve relatively good results. The approach is evaluated on the TGRL specification of a Virtual Interior Designer application GRL model, including 42 requirements covering both security and non-security aspects. The metrics used for evaluation include precision, recall and F1-score, however, since true negatives can be extremely more numerous compared to true positives and false negatives, they are typically not used directly in the calculation of precision and recall.

The model achieves high performance on all metrics, and accurately identifies relevant links (TP and FN), but it also predictably misses some, that, upon further interaction with the LLM agent, are identified as valid, showing the limitation of LLMs in explainability, transparency, and consistency in the outputs. Limitations of the approach include the generation of duplicate links, stemming from the presence of similar goals or requirements, but also the quality of the prompt, as further refinement and use of few-shot examples may improve performance. Also, evaluation should be extended to more and larger application examples and real-world case studies, while also incorporating different types of requirements (e.g. required, optional, recommended, and prohibited requirements) with varying structures and formats (including, for example, user stories), and implementation with other available LLMs to generalize the approach's applicability.

Traditional traceability link recovery techniques heavily rely on manual feature engineering and lack deep semantic understanding, while more recent ML and DL approaches improved on many aspects but challenges remain in handling limited labeled training data, large-scale scenarios and multilingual contexts, multi-type and cross-level requirements traceability links. RQ4 reviews the most recent approaches leveraging pre-trained and large language models, highlighting the relevant enhancements introduced.

T-BERT[32] uses BERT with multi-stage training to recover trace links between NL and PL artifacts, by leveraging transfer learning from the auxiliary code search task, which significantly improves traceability performance. Three architectural variants of BERT are tested across three open-source projects and the results show that they all outperform traditional IR models and deep learning RNN approach. MT-BERT[33] handles bilingual traceability by directly processing multilingual artifacts with BERT without the need for translation in pre-processing, and uses pre-existing trace links for fine-tuning, enhancing the model's adaptability in industrial environments. MT-BERT is evaluated on 14 Chinese-English bilingual software projects and achieves superior performance, particularly in larger projects, than the best-performing information retrieval model GVSM with mono-lingual word embeddings. *Xia and Wang*[34] propose a knowledge-graph approach using BERT+BiLSTM+CRF to extract entities from software documents and create entity relationships using knowledge graphs. The method relies on graph-based similarity scoring though graph complex theory to obtain the result of requirement traceability. The model is evaluated on real-world software requirement documents, where it achieves a high average F1 score for named entity recognition and demonstrates high traceability accuracy especially as the number of entries increases. DRAFT[35] addresses cross-level requirement tracing by combining project-specific second-phase pre-training, along with process features and extended features leveraging historical trace links. The evaluation reveals considerably better results compared to baselines such as VSM, relevance feedback, and TraceBERT. *Majidzadeh et al.*[36] use CodeBERT with code augmentation techniques and multi-stage fine-tuning to improve multi-type requirements traceability link recovery, including issue-commit, issue-method, and documentation-method links. Experiments on extensive 14 Java open-source projects demonstrate that the proposed method improves over both T-BERT and LSI-based method CODFREL, with code augmentation significantly enhancing doc-method and issue-commit link recovery and 3-stage fine-tuning particularly improving issue-method link recovery. BTLink[37] focuses on issue-commit link recovery by combining RoBERTa and CodeBERT, using semantic features from NL and PL data, and leveraging the models' transfer learning capabilities for enhanced performance. BTLink is evaluated on labeled link datasets constructed from open-source software and exhibits better performance in both within-project and cross-project contexts than state-of-the-art approaches like DeepLink, FRLink and hybrid-linker and variants of BTLink using traditional ML models as classifier layer. EALink[74] introduces knowledge distillation to compress a large pre-trained teacher model into a lighter student model to reduce computational overhead while maintaining accuracy in issue-commit link recovery. EALink uses contrastive learning to fine-tune the model for issue-commit link recovery in a multi-task way and addresses the issue of conflicting false links with a link generation mechanism for constructing reasonable false links to be used for training. Evaluation on a newly constructed, large dataset reveals the superior performance of EALink compared to state-of-the-art methods including T-BERT, DeepLink and VSM on large projects. MTLink[38] builds on this concept by leveraging multi-teacher knowledge distillation combining CodeBERT and GraphCodeBERT to allow building a compressed model without incurring in a significant loss of information, and integrates adaptive multi-task learning to train the model on different traceability tasks while alleviating the issue of class imbalance. MTLink is compared to EALink, T-BERT, DeepLink, VSM, and shows improvements in terms of several performance metrics and especially in large-scale datasets. DSSLink[39] is focused on handling the problem of labeled training data scarcity in issue-commit link recovery, by introducing semi-supervised learning techniques and pseudo-labeling. The effectiveness of DSSLink is verified on datasets from fifteen open-source software projects, with improved performance in most cases compared to T-BERT and BTLink. Finally, studies like [40, 41] leverage LLMs like Claude and GPT-3.5-turbo through their conversational interfaces to generate traceability links, offering an innovative way to enhance traceability without the need for extensive model pre-training. In conclusion, LLM-based methods along with advanced techniques like knowledge distillation, multi-task learning, and semi-supervised learning, offer substantial improvements for requirements traceability, especially when addressing challenges such as data scarcity, bilingual contexts, and variability in traceability links, significantly enhancing scalability and efficiency even in large and complex software projects.

3.2.5 RQ5: What is the potential role of LLMs in automating API recommendation?

Application Programming Interfaces (APIs) provided by software libraries or frameworks are essential in modern software development. Due to the huge number of available APIs, many approaches have been proposed over the years to help developers find suitable APIs for their needs. API recommendation approaches can be mainly distinguished into two categories [81]:

- Query-based approaches: they generate API recommendations based on a query provided by the developer, which could be a natural language description, keyword, or other input formats.
- Code-based approaches: they focus on predicting the next API based on the code context surrounding the point of prediction.

  Within these two categories, the proposed methods have evolved from basic information retrieval- (BIKER[82], RACK[83]) and pattern-based approaches to more sophisticated learning-based models that leverage deep learning (DeepAPI[84]; APIHelper[85] for code-based).

The main limitations encountered by these approaches are their limited contextual understanding, generalization and transferability between different API ecosystems or code programming languages, while the main challenges to address include the quality of queries; the quality of code before the recommendation point; user-defined APIs; cross-domain adaptation; the complexity of API sequences; contextual understanding; and, in case of learning-based methods, the reliance on a vast amount of training data. All these factors contribute to making recommendation of suitable APIs a hard task for developers. Following the recent advancements in learning-based technology, API recommendation research has started leveraging pre-trained language models and transformer-based components. RQ5 explores the main approaches in this field.

CLEAR by *Wei et al.*[42] leverages BERT-based sentence embedding and re-ranking to obtain suitable API recommendations based on queries and Stack Overflow (SO) posts. The goal is to tackle the difficulty of IR-based solutions at capturing the semantic-related sequential information, and the difficulty that both IR- and neural-based approaches face at grasping the semantic difference among lexically similar queries. The model used for both tasks is RoBERTa. CLEAR's BERT-based model embeds the whole sentence of queries and SO posts text descriptions rather than combining the embeddings of each word as in bag-of-words approaches, thereby preserving the semantic-related sequential information.

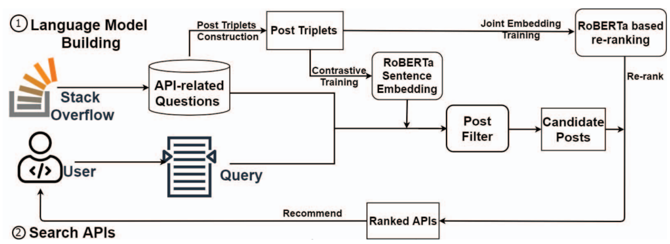


Figure 3.46: CLEAR overview.

The components of the framework include a BERT-based embedding model, a filter, and a reranking model. The BERT sentence embedding model is trained with contrastive learning [86] to learn semantically equivalent representations of queries or SO posts, regardless of their lexical information. Contrastive training takes as input triplets composed of an input query, a positive sample post that is semantically equivalent to the query, and a negative sample post that is not related to either of the above. In the contrastive learning architecture, RoBERTa is used as base model for sentence embedding, followed by a pooling layer that connects it to a triple deep neural network layer for feature extraction of input sentences, and, finally, to the loss function. The training goal is to minimize the distance between similar sentences and maximize the distance between unrelated sentences, learning whether two sentences are semantically similar regardless of their lexical similarity. The result is a selection of candidate SO posts.

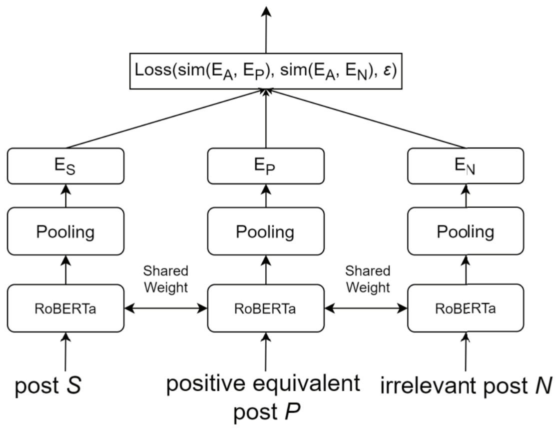


Figure 3.47: Architecture for contrastively training RoBERTa based sentence embedding model.

CLEAR further filters relevant SO posts for a given query and then leverages RoBERTa fine-tuned for classification with joint embedding training to re-rank candidate SO posts. In particular, the reranking model computes the probability that a query and a given candidate post have the same label (where the label is whether two posts have the same APIs). Based on these probability scores, the model ranks the selected candidate posts and finally recommends the APIs from the top-ranked SO posts for the query.

The authors evaluate CLEAR against state-of-the-art approaches including BIKER, RACK, and DeepAPI, but also a CLEAR variant that separates the filter from the reranker with the simple pre-trained RoBERTa model to explore the effect of contrastive learning on performance. The experimental results on three different test datasets confirm the effectiveness of CLEAR for both method-level and class-level API recommendation and the importance of implementing contrastive learning in the framework. Compared to the state-of-the-art API recommendation approaches, CLEAR improves the MAP by a range of 25%-187% at method-level and 10%-100% at class-level.

Differently from previous approaches for API recommendation that leverage either natural language text (query-based approaches) or source code context (code-based approaches), PICASO by *Irsan et al.*[43] aims at improving API sequence recommendation by combining the two types of input, specifically code comments from GitHub and crowd-source knowledge from Stack Overflow posts. The goal is to address the problem posed by the large volume of available APIs, where multiple APIs can be used to implement the same functionality and developers have to choose the most suitable for their needs.

PICASO is a two-phase framework: given a developer's query in the form of a code comment or annotation, the model initially identifies the SO post that is most semantically similar to it. For this purpose, the authors adapt CLEAR's framework in such a way that, instead of leveraging sentence embedding to map a SO post to other similar SO posts, PICASO utilizes it to map an annotation towards similar SO posts. The sentence embedding filtering model trained via contrastive learning produces embeddings for the annotation and SO posts' titles, and based on the cosine similarity between these embeddings, the top-10 most relevant SO posts are considered. After that, the re-ranking classification model trained via joint embedding finds the SO post that is the most semantically relevant to the given annotation. The second phase involves expanding the query with the title of the relevant SO post and APIs mentioned in its answers and using the extended queries to fine-tune the CodeBERT model for API sequence generation. Specifically, the title and set of APIs mentioned by the previously retrieved post are extracted from it, encoded by CodeBERT along with the annotation and concatenated into a single feature vector. This then is passed to the decoder, which generates the API sequence by sequentially predicting output tokens based on the input and previously generated output tokens.

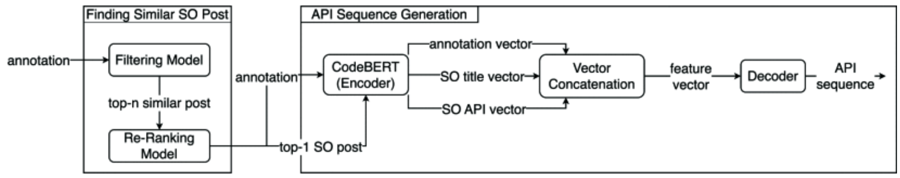


Figure 3.48: PICASO architecture.

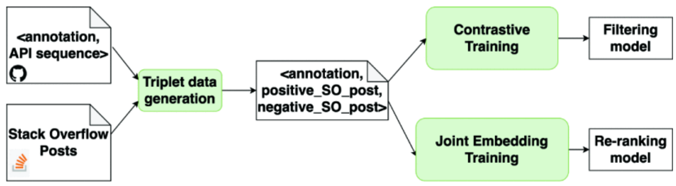


Figure 3.49: The training process for finding the most similar Stack Overflow post.

By integrating the additional context provided by Stack Overflow, PICASO helps in cases where the original query is too vague or lacks specificity. For example, the system can infer additional information about a query's intent by linking it to a more complete description from Stack Overflow, resulting in more accurate API suggestions.

The model is evaluated on a dataset derived from GitHub and Stack Overflow, and the experiments validate PICASO's effectiveness, showing that query expansion with both the title and API set from SO posts contributes to enhancing the API sequence generation compared to only using code annotations or expanding with the title alone. Moreover, the results show that PICASO outperforms the baseline model CodeBERT, with an improvement of more than 10% on the BLEU-4 score.

Besides query expansion, another technique that research adopts to improve the query-based API recommendation is query clarification. This technique involves asking clarifying questions to the user and providing options in order to refine the query statement. Deep learning-based query clarification methods exhibit superior ability at expressing the user's intent, but still face many challenges: they rely on highly diverse and vast training data, suffer from out-of-vocabulary (OOV) failures when the correct extension keywords are not present in the training dataset, require multiple rounds of question clarification to accurately capture the user's intent and obtain the correct extension keywords, and do not account for the relationship between clarification rounds.

Knowledge Graph (KG)-based methods [87] improve query clarification thanks to the large number of APIs contained in a KG, along with their corresponding clarification information in the form of aspects (e.g., purpose, type, and event) and options. Still, OOV failures can occur when queries fall outside the KG's scope, and the reliance on pre-designed templates brings rigidity and hinders fluency in the conversations. *Huang et al.*[44] address these limitations, proposing a novel knowledge-guided query clarification approach for API recommendation that leverages an LLM guided by KG as a neural knowledge base to overcome OOV failures and generate fluent and appropriate clarification questions and options. At the same time, the structured API knowledge and entity relationships stored in the KG help to filter out noise in the process of clarification question and option generation that may easily arise from the inherent randomness present in LLMs outputs and their broad knowledge base. This allows to transfer the optimal clarification path from KG to the LLM, increasing the efficiency of the clarification process and control on the output.

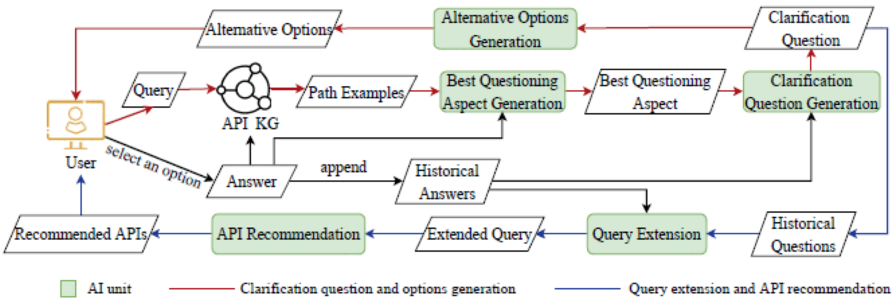


Figure 3.50: Overall architecture design.

The LLM used is GPT-3.5, although any model with in-context learning capability is applicable. The approach is designed as a chain of separate LLM prompts, each addressing a different step, like Clarification Question Generation; Alternative Option Generation; Query Extension (leveraging the historical questions and answers from the conversation); API Recommendation, based on the extended statement.

The study explores the performance of the model against two query clarification baselines (a deep learning-based method and a KG-based method) on two datasets (one where the query statements are present in the KG and the other where they are not). The proposed approach shows a significant improvement in MRR and MAP, with the best results in the former case, showing its efficiency in addressing OOV failures in query clarification and identifying user intent for API recommendation. Additionally, ablation experiments reveal that the KG guidance and the knowledge-guided pathfinding strategy are crucial for the performance of the model, as they respectively help in reducing noise and enhancing efficiency and controllability.

The approach demonstrates a way to take advantage of the strengths of both KG and LLMs, while effectively compensating for their weaknesses, achieving high precision, robustness, and controllability, accurately generating clarification questions and extending queries aligned with user intent.

While these studies mainly model API recommendation as a recommendation task, *Gu et al.*[45] propose a novel API recommendation framework named APICOM based on the observation

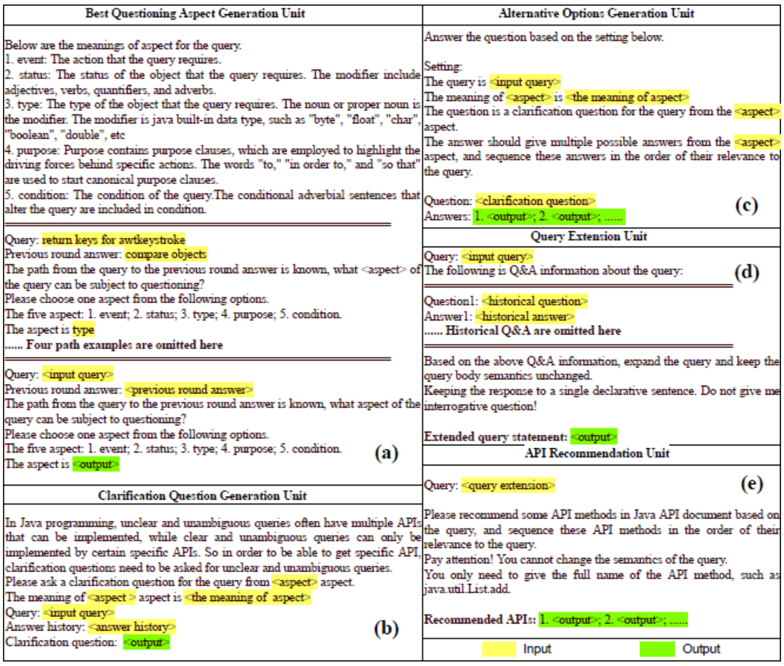


Figure 3.51: Prompt Design for the Five AI Units.

that developers generally remember API prefixes, and rather need help for the remaining part of the API. The authors thus propose to model the API recommendation problem as an automatic completion task, where developers input their query statement and known API prefixes, and the corresponding correct API is generated based on this data.

The data preparation for the API completion task involves generating prompts that reflect real-world cases where developers know only part of an API and need help completing it. Random masking is used to create "prefix prompts" where words at the end of an API are masked, mimicking how developers construct APIs and multiple versions of incomplete APIs are created for each API. These generated prompts and the original query are taken as input for a novel adversarial training model for data augmentation, named ATCom. For each input, ATCom generates adversarial examples to augment the training set. This data augmentation process addresses the problem posed by the dependency on the quality of training data, which is time-consuming and labor-intensive to collect manually. The generated adversarial examples are used alongside the original inputs as training data to fine-tune CodeT5 to predict the missing section of APIs. The encoder of CodeT5 processes the input to generate a hidden state vector, that the decoder uses to generate API tokens through a softmax layer. This way the model learns API completion patterns.

The performance of APICom is evaluated on an extensive corpus of queries and corresponding APIs, and the baselines for comparison include classical API recommendation approaches (i.e. BIKER, RACK, and CLEAR) but also standalone pre-trained models (i.e. CodeBERT, UniXcoder and PLBART). APICom is found to outperform all baselines in terms of the performance measures Exact Match@1, MRR, and MAP. Finally, ablation studies on the use of prompt learning and considering several variants for the adversarial training and the pre-trained model confirm the effectiveness of the chosen components. These results indicate the improvement in semantic diversity and richness of the generated APIs thanks to data augmentation and the effectiveness

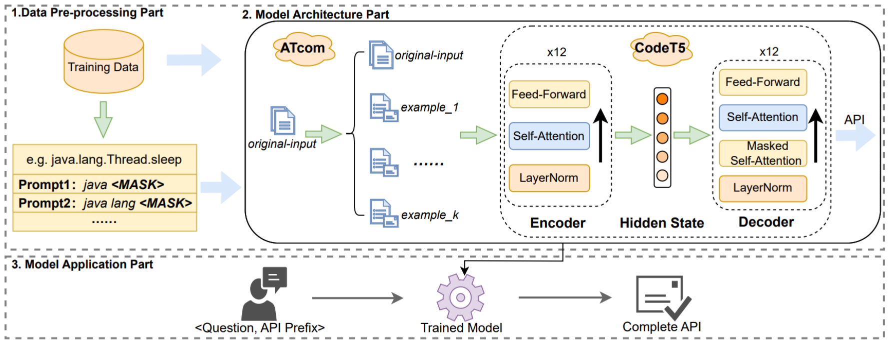


Figure 3.52: APICOM framework.

in recommending APIs that are better aligned with developers' intent thanks to using prefix prompts.

In the context of Mashup applications development, where multiple open APIs or remote services are combined into a lightweight service composition application, API recommendation systems face the same challenge of finding suitable and high-quality APIs given their huge number and unstructured NL formulation. This is also given by the fact that the modeling and representation of Mashup requirements and clusters are often ignored, and recommendation results lack proper diversity, as they usually only provide the name of recommended open APIs and do not include related information such as the popularity of the open APIs and functional similarity between the open APIs and Mashup requirements.

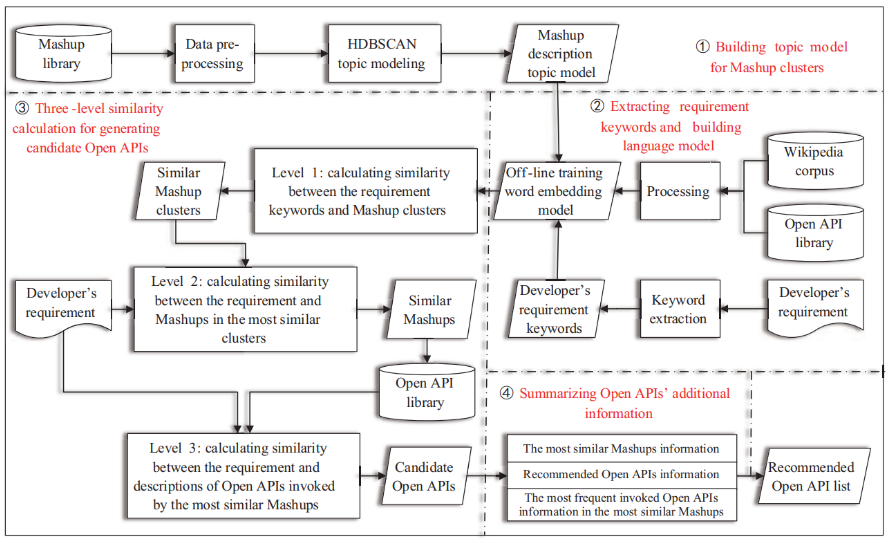


Figure 3.53: DLOAR framework.

Wang Y et al.[46] propose a deep learning-based open API recommendation (DLOAR) approach to improve the accuracy, efficiency and relevance of the recommendation results in mashup environments. The method takes developers' NL-based requirements, Mashup descriptions, and open API descriptions as inputs and processes them at different levels. First, it uses RoBERTa to model Mashups by extracting high-quality word and sentence embeddings with contextual information from Mashup descriptions, thus overcoming the input limitations and semantic loss issue of traditional approaches. Next, the model uses hierarchical density-based spatial clustering of applications with noise (HDBSCAN) to construct topic models, aggregating Mashups with similar semantics into clusters, and extracting topic words. DLOAR then processes complicated NL-based requirements by extracting keywords from them and builds the language model for Mashup requirements to obtain the clusters that are most relevant to Mashup requirements, by comparing the similarity between topic words of Mashup descriptions and keywords of developers' requirements.

Following this and based on the requirement keywords, a neural network model computes the similarity at the cluster level, Mashup level, and open API level, to find the most relevant open API(s). Finally, along with the recommended APIs, DLOAR also provides additional useful information (such as similar Mashup descriptions and the invocation frequency of the most matched open API), to improve the diversity of results and help developers make more accurate choices. Although the approach requires manual tuning of multiple hyperparameters and can be improved in the explanation of recommended open APIs, the evaluation shows that it outperforms commonly used approaches (leveraging TF-IDF, LDA, Word2Vec, and Sentence-BERT) in precision, recall, F1-measure, MAP and MRR. This reveals the benefits of HDBSCAN and three-level similarity calculation in improving clustering accuracy and recommendation relevance within the framework, which are also further confirmed through ablation studies.

One limitation of most existing works in recommendation systems is that they use a single source of information (Deep-Learning-based approaches focus on using textual features, while knowledge graph-based focus on using graph structure features), and do not fully consider the two, which may lead to suboptimal representation and damage recommendation performance.

C. Zhang et al.[47] propose a Cooperative Mashup Embedding (CME) neural framework that integrates knowledge graph embedding and text encoding.

First, a graph structure is constructed to represent the relationships between the two entities of interest, i.e. mashups and APIs, and then the Node2Vec embedding algorithm is employed to convert these entities into numerical vectors to be used for downstream tasks like recommendation. Meanwhile, BERT is used to encode the textual description of Mashups into vector representations and build a classification model where API recommendation is modeled and solved as a multi-label classification task. BERT encodes each word in a sentence into Token, Segment, and Position Embeddings. The text is processed into subwords, with the addition of [CLS] and [SEP] tags for classification, and an embedding vector is acquired for each subword. Specifically, the [CLS] vector represents the whole sentence and is used for categorization.

To enhance feature selection and facilitate interaction between the text and the knowledge graph domain, three fully connected layers are used within the BERT component: one is a linear transformation that serves as a pooling mechanism for feature stability and overfitting prevention; the other two perform nonlinear transformations with tanh and sigmoid activation functions to enable BERT to learn and execute more complex tasks.

To enable knowledge and feature exchange, two decoders are incorporated to decode the output of the BERT encoder into distinct semantic spaces, i.e. the knowledge graph embedding space and the API-specific space. This design enables the framework to simultaneously capture Mashup-API calling patterns, Mashup/APIs co-occurrence patterns, and textual information of mashups, thereby enriching the representation of mashups and enhancing the accuracy of Web

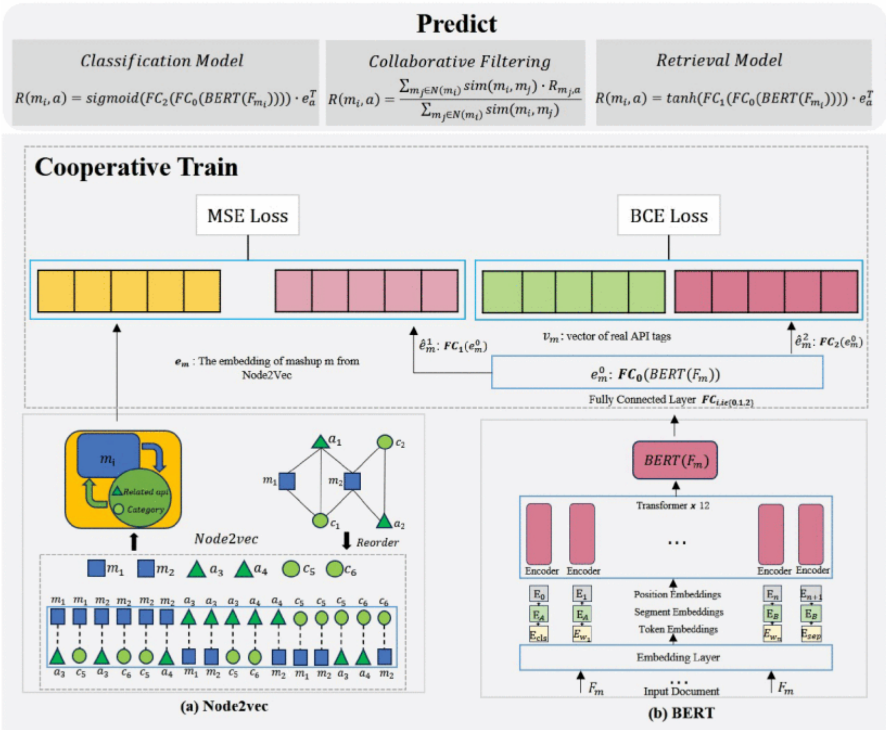


Figure 3.54: CME framework. Node2Vec realizes knowledge graph embedding, and BERT fulfills the text embedding. Two modules will exchange and share knowledge from two embedding spaces in cooperative training. Mashups are represented by rectangles, APIs are represented by triangles, other entities are represented by circles, and edges represent entity relationships.

API recommendations. During training, the model uses an alternating collaborative training strategy to align the deep representations from BERT with the shallow representations from Node2Vec through repeated iterations until convergence, and thus better integrate both text and graphical data for greater recommendation accuracy.

The diverse representations of entities generated by the CME model are leveraged to construct three recommendation algorithms that take the textual descriptions of Mashups as input and recommend Web APIs based on their association strength with the input. These models include: the CLASSIFICATION MODEL (CM), that treats the API recommendation task as a multi-label classification task with the text-based representation of a mashup as input, the Web APIs as labels for training and the probability distribution of all Web APIs as output, generating a sorted recommendation list; the RETRIEVAL MODEL (RM), that treats the task as a retrieval process, assuming both the mashup requirement and the candidate APIs are expressed in the same semantic space, and outputs a recommendation list based on the similarity between the mashup and candidate APIs; finally, COLLABORATIVE FILTERING (CF), that assumes mashups with similar requirements are more likely to reuse similar combinations of Web APIs and thus defines a set of mashups that are the most similar to a new input mashup, based on their text-based representation, and computes the relevance score between the mashup and an API by aggregating all relevance scores from the set of mashups to a specific API. The recommendation list is generated by sorting all candidate APIs on their relevance to the mashup.

Experimental results on the ProgrammableWeb dataset indicate that the proposed method achieves better performance in metrics including precision, recall and NDCG (Normalized Discounted Cumulative Gain) over representative recommendation methods covering topic modeling, random walks, and deep learning. The best overall performing model in particular is the Cooperative Mashup Embedding Classification Model.

Regarding API recommendation for the Python programming language, *K.Li et al.*[48] propose the code-based method PyBartRec to overcome the limitations of previous query-based and code-based approaches.

Query-based API recommendation techniques for Python need to construct a Python knowledge base using data sources such as StackOverflow, and then encode user queries as vectors to find the corresponding APIs within the knowledge base. In order to select the desired API, developers usually need to understand the code context which increases the effort and difficulty in making accurate predictions.

Code-based API recommendation methods, on the other hand, take into consideration the code context preceding the recommendation point and make recommendations based on high-quality local data flow information provided by data-flow analysis tools. However, these tools are designed for static languages and heavily rely on the accuracy of type inference, which is difficult for dynamic programming languages like Python. Thus, using them in these scenarios the data flow obtained from these tools is often conservative and leads to erroneous dependencies in the data flow, resulting in many false positives API recommendations.

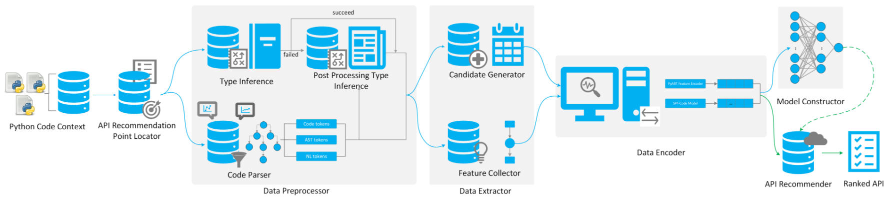


Figure 3.55: PyBartRec framework.

To address the type inference issue, Deep-Learning-based models have been proposed recently, but they struggle with user-defined types. PyART[88], a state-of-the-art Python API recommendation method, collects all possible candidate types and their corresponding APIs in the same project in cases of type inference failure, thus generating an excess of candidate APIs and negatively affecting the accuracy of recommendations. Moreover, the approach adopts data flow information to model the API and its surrounding syntactic entities and semantic properties, with an optimistic analysis method to address the issue of error dependency caused by conservative data flow. The resulting optimistic data flow however only includes the local information around the APIs and if the information is not indicative of the correct APIs, it often leads to incorrect recommendation results. PyBartRec addresses the difficulty in type inference and the reliance on optimistic data flow information utilizing a Transformer-based code pre-trained model that uses code structure as input, SPT-Code[89]. Instead of analyzing the data flow information only, the model extracts the overall semantic features of Python code snippets, thus compensating for the insufficient local information in the optimistic data flow and allowing recommending correct APIs even when they are not included in the local data flow information. This semantic information is also leveraged to perform a post-processing type inference in scenarios of type inference failure, narrowing down the range of candidate types.

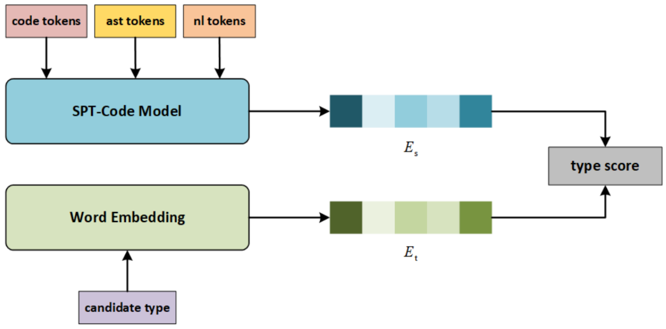


Figure 3.56: The general framework of post-processing type inference.

PyBartRec begins by extracting the Python Code Context preceding each recommendation point in a Python source code file and processing it into a sequence of features.

The API Recommendation Point Locator identifies recommendation points; the Data Preprocessor processes each context before a recommendation point with a lightweight Python type inference tool that performs static type inference on the caller object of the target API. If type inference is successful, all callable methods of the inferred type are gathered as candidate APIs, otherwise, the Post-Processing Type Inference method is used on the caller object.

The input for the Post-Processing Type Inference component is represented by code tokens, AST tokens, and NL tokens obtained by parsing the context before the current recommendation point. These are used to obtain the vector representation of the semantic information of the entire code. Also, all the imported packages in the current source code file are collected as potential candidate types, including user-defined types and some built-in types. The type list is then encoded into feature vectors using word embedding and the distance between vectors is computed to obtain the semantic similarity and derive the probability of each type in the list being the correct type required in the present context and the list of correct types is generated, extracting all callable methods of these types as candidate APIs. The same tokens obtained by the Code Parser are also input to the Data Extractor, to extract features from them, mainly including optimistic data flows, token similarity in data flows, and token co-occurrence. These, along with the initial code, AST, and NL tokens obtained by the Code Parser, are then encoded into feature vectors, one for each candidate API, by the Data Encoder. Specifically, the code, AST, and NL tokens representing the contextual semantic information are encoded by the SPT-Code model. Finally, the Model Constructor combines the feature vectors with the positive and negative candidate APIs to form positive and negative samples to be used for training a deep neural network, and the API Recommender utilizes the trained model to compute the scores for each candidate API and then ranks them accordingly.

PyBartRec is evaluated in eight popular Python projects and the experimental results show that the approach outperforms the two state-of-the-art solutions PyART and Visual Studio Intellicode in both intra-project and cross-project recommendation, achieving a 16%-18% higher top-k accuracy in intra-project API recommendation, and a 21% higher MRR in cross-project API recommendation, compared to PyART, demonstrating its effectiveness.

Z.Li et al.[49] introduce a novel framework, PTM-APIRec, to tackle the main challenges faced by existing pattern-based, traditional ML-based or DL-based methods for API recommendation: the need for large training datasets, the dependency on manual feature extraction, and also the

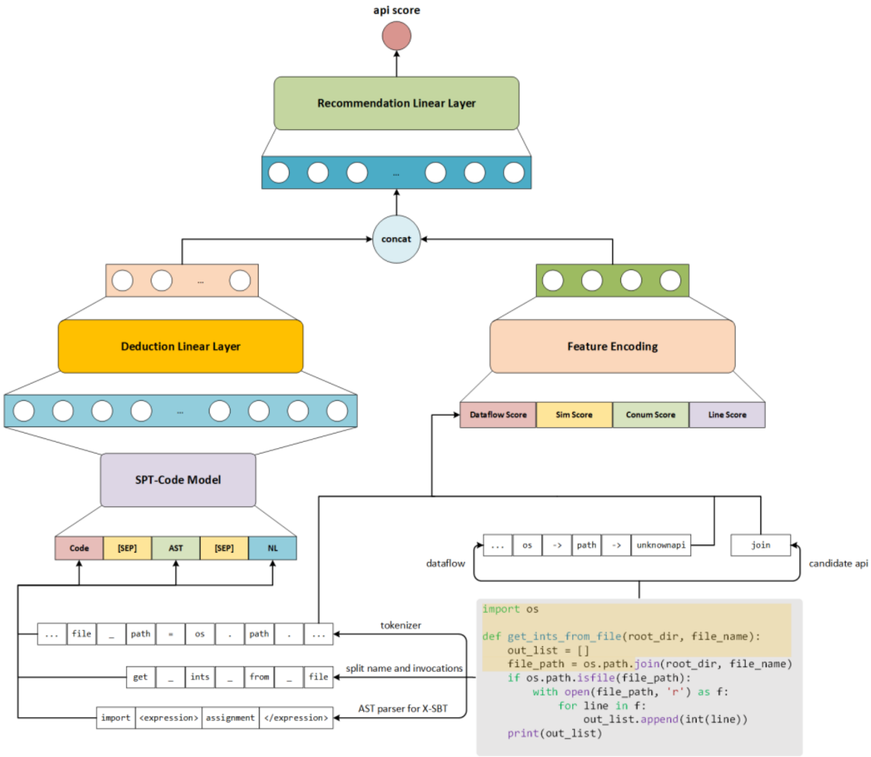


Figure 3.57: PyBartRec structure.

struggle to fully capture the semantic richness of code beyond API sequences, which all lead to suboptimal recommendation performance. Moreover, recent techniques leveraging pre-trained models, such as APIRecX[90], use dedicated models based on API sequences rather than general code pre-trained models, mainly because the API recommendation task considers an API as a token, whereas general pre-trained models have a different tokenization approach e.g. BPE. On the other hand, directly using the vocabulary of the general pre-trained model means that the API recommendation task is implemented as a general code completion task and only one sub-token of the candidate API can be generated at a time. However, since API names are fixed, such a strategy may result in predicting APIs that do not exist.

PTM-APIRec introduces several advantages: leverages general code pre-trained models to fully encode the input code snippets considering contextual information from the entire input code rather than limiting itself to API sequences; requires less training data, as it leverages code pre-trained models that encode general programming knowledge from large, unlabeled code corpora, which are easier to obtain than specialized API recommendation datasets.

Moreover, PTM-APIRec uses separate general vocabularies and API-specific vocabularies for the APIs, to ensure that the input code's semantics are fully captured and only valid APIs are recommended. The approach builds API vocabularies from the source code by gathering all the API calls through a static analyzer that navigates the source code. The API calls include the signature and class details. If an API call is not already present, it is added to the vocabulary, and this process is repeated until all the APIs in the training set are collected. After identifying the object's type at the hole token, all APIs associated with that type are retrieved as candidate APIs. If the type information at the hole position cannot be directly extracted, the system retrieves APIs for any types previously encountered before the hole position. A type is considered valid if a prior API call has returned an object of that type. For static methods, which do not require an object, these methods are added to the candidate list even if the type is unavailable. PTM-APIRec can be extended to new API libraries by first preparing the corresponding API vocabulary and then training the model on the corresponding dataset. Thus, adding a new vocabulary does not affect other vocabulary embedding and the model does not need to be re-trained from scratch. This makes PTM-APIRec suitable for evolving software environments.

The two key innovations of the approach are that it utilizes the embedding of multiple special [MASK] tokens as the context embedding of the APIs to be predicted, instead of the [CLS] token; and that instead of treating the API recommendation task as a multi-label classification or generation task, it models it into a matching problem by adopting pre-trained models to encode both the API context and the API candidates.

After tokenization, the Encoder uses general code pre-trained models to encode the complete code snippet in the method body that appears before the API to be predicted, in order to extract features and obtain a contextual vector representation. The Analyzer extracts class information of the objects in the code snippet to narrow down API candidates (in most programming languages that support object-oriented programming, an API is called by an object, therefore the type and class of the object before the hole token define what APIs can be called). Moreover, previous API calls and their relative positional relations should also be considered, since in some cases an object should only call a certain API once (e.g., opening and closing a file). The Retriever retrieves API candidates, considering all APIs of the type obtained by the Analyzer, and an embedding layer encodes them into vectors. PTM-APIRec treats API names as text sequences, tokenizes them with the pre-trained tokenizer, and uses the last hidden state of the [CLS] token as the API's embedding. These embeddings are generated for each API in the vocabulary, and combined to build an initial embedding layer, later fine-tuned during training. Leveraging the pre-trained models to generate initial weights for the embedding layer speeds up the embedding layer learning. For models like CodeGPT, which lack an encoder, the embedding layer is randomly initialized. All candidate APIs are transformed into vectors and concatenated into a matrix for further processing. Subsequently, PTM-APIRec computes the similarity score between the context feature vector and each candidate vector using the dot product, which makes the model highly flexible as it allows new APIs to be easily incorporated by updating the embedding layer without the need to re-train the entire model. Finally, by applying the softmax function to the similarity scores received as input, the Predictor generates the list of APIs ranked according to their probabilities of being the correct recommendation.

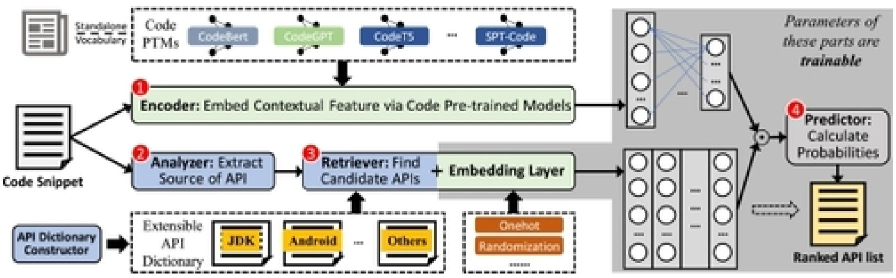


Figure 3.58: PTM-APIRec overview.

PTM-APIRec can adapt to different types of pre-trained models, each with its own strategy to encode the code's context. When using the encoder-only model CodeBERT trained for the mask prediction task, the token that needs recommendation (called hole token) is replaced with a mask token. After feeding the code into the pre-trained model, the last hidden states are extracted and the vector at the mask's position is taken as the context vector. If the API name requires multiple tokens, multiple mask tokens are used, and their average hidden state values are taken as the context vector. Then, based on the context vector, the model's head derives the probability of each word in the vocabulary being the correct prediction.

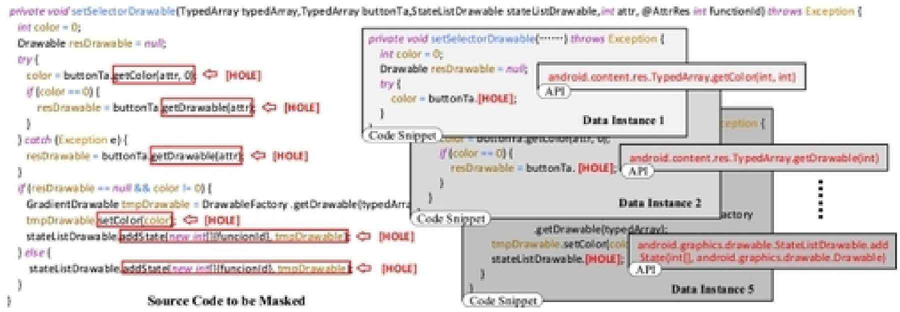


Figure 3.59: Example of masking Source Code and generated data instances.

When using the decoder-only model CodeGPT, each input word is passed through a language model head to get the probability of the next word, and the last hidden state of the token preceding the hole token is taken as the context vector. When using the encoder-decoder model CodeT5, the last hidden states of the decoder's output at the hole token's position are used as the context vector. Finally, similar to CodeBERT, UniXcoder uses a special mask token for code completion tasks. The mask token is placed at the hole position, and the last hidden states at that position are used to form the context vector, but, unlike CodeBERT, the mask token can be used to predict multiple tokens, so only one mask token is needed even when the API name requires to be split into multiple tokens.

The approach is evaluated on code-based data from the APIBench dataset, focusing on the Java API and Android API, against code completion approaches based on code pre-trained models (CodeBERT, CodeGPT, CodeT5, UniXcoder) and state-of-the-art API recommendation approaches (APIRecX, N-gram models and APIhelper). PTM-APIRec outperforms all baselines on both top-k and MRR metrics, especially when incorporating models with the encoder component, like CodeBERT, CodeT5 and UniXcoder, as they can encode candidate APIs to obtain initial embeddings, which improves accuracy.

In particular, experimental results show that PTM-APIRec demonstrates a key advantage in predicting APIs with more tokens compared to models like CodeGPT and CodeT5, and thus in handling more complex predictions. Moreover, while the top-1 accuracy is high in code completion approaches, there is limited improvement in top-5 and top-10 accuracy due to the difficulty of predicting longer APIs. For comparison with the API recommendation approaches PTM-APIRec outperforms APIRecX in specific contexts, succeeding in predicting APIs that APIRecX fails at, especially when there is a need for complex context comprehension, such as handling API sequences that involve execution order, assignment operations, and handling keywords effectively, thanks to using source code directly as input rather than just API sequences. Another result is that using pre-trained models with an encoder (such as CodeBERT, CodeT5, and UniXcoder) for PTM-APIRec leads to better performance because they can encode candidate APIs to obtain initial embeddings, which improves accuracy. However, PTM-APIRec also has limitations. It can miss out on correctly predicting subsequent API calls if it fails to learn from earlier predictions in a sequence. This gap in learning from prior API calls was noted in certain failure cases, where APIRecX successfully predicted the remaining API based on prior context, whereas PTM-APIRec continued making incorrect recommendations.

The evaluation also demonstrates the high performance achieved by PTM-APIRec even when the size of the training data is small, thanks to the use of general pre-trained models, and the quick adaptation to new API libraries with only a small amount of training data or even without training for some APIs. This is due to the embedding strategy. When importing a new library, the new API vocabulary is collected and an initial embedding for each API is obtained through pre-trained models. Since the embeddings of the new APIs are distinct from the original library, the model can be easily applied to the new library one affecting the original one and correctly recommend some APIs even without training. Beside the longer inference time compared to models with simpler structures like n-gram and APIHelper, the model narrows down API candidates based on object types, making it more effective for static languages like Java, where object type information is more readily available, but less so in dynamic languages like Python. Additionally, considering the complete code as input to encode the context, as opposed to specializing the API calls, may cause the problem of ignoring the previous API call information.

In summary, PTM-APIRec advances API recommendation by overcoming the limitations of previous approaches, offering a more robust, adaptable, and efficient solution that leverages the power of pre-trained models to improve accuracy and scalability in real-world coding environments.

Considering library-based code generation with LLMs, existing API recommendation methods have unsatisfactory performance, especially when the code to be generated imports libraries that are not present or rarely present in the training data of the models. The main challenges reside in the scarcity of training data and the granularity inconsistency between requirements and APIs.

Ma and An, et al.[50] propose CAPIR (Compositional API Recommendation), to alleviate the need for a large amount of training data and facilitate API recommendation for coarse-grained requirements that require a combination of multiple fine-grained APIs.

The framework involves leveraging an LLM-based Decomposer to break down coarse-grained user requirements into a fine-grained subtask sequence at the granularity level of APIs in the target library, followed by an embedding-based Retriever to identify relevant APIs corresponding to each subtask from API documentation and finally an LLM-based Reranker to filter out redundant APIs and provide the final recommendation.

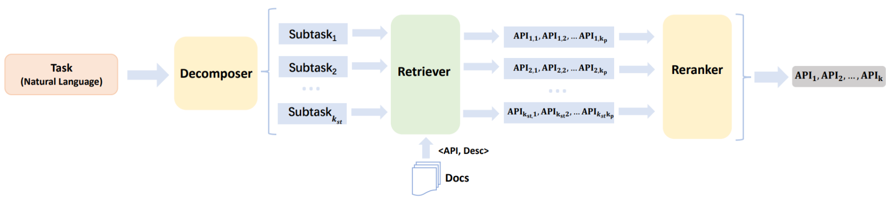


Figure 3.60: Overview of CAPIR.

To guide the LLM in accurately decomposing requirements' descriptions, the prompts for the Decomposer include summarizations of API usage code examples extracted from development documents as few-shot examples. To construct these examples, CAPIR first crawls API documentation of a library and extracts code snippets that invoke multiple APIs. Then, it leverages an LLM to generate the code summarization and the functional descriptions of APIs called by the code. The summarization is treated as the high-level task and the functionality descriptions as the API-level subtasks.

Due to the input length limitation of LLMs, for a given task, CAPIR uses an embedding-based Selector to only take the most relevant examples to be added into the prompt, based on the cosine similarity between the feature embedding of the task description and of the code summarization. While previous approaches fine-tuned the embedding model using training data, CAPIR employs an off-the-shelf embedding model to deal with low-resource libraries, which also streamlines the process.

To retrieve the APIs from the API documentation CAPIR utilizes the same off-the-shelf embedding model used as Selector, since fine-tuning a library-specific embedding model is not feasible with low-resource libraries. Again, the Retriever computes the cosine similarity between the embeddings calculated for the subtask and the API description and returns the most similar APIs as candidates for the subtask.

Subsequently, the LLM-based API reranking module first reranks the candidate APIs within subtasks and then across subtasks, leading to the final recommendation results. This cross-subtask approach helps in cases where API and subtask granularity is not perfectly aligned, hence multiple APIs or, on the contrary, no APIs, may be associated with a single subtask.

The Summarizer, Decomposer, and Reranker modules are implemented using GPT-3.5-turbo, while the Selector and Retriever use the ada-embedding-002[91] model.

The study compares CAPIR with baselines on API sequence recommendation and library-oriented code generation for both single-library and multi-libraries and conducts an ablation study to assess the impact of the examples in the Decomposer prompt, the impact of the Decomposer itself (by removing it and directly using the original task to retrieve APIs from the documentation), and that of the Reranker (by seeing how the model performs when directly returning the top-k retrieved results). Finally, the evaluation also includes testing the performance on API recommendation when replacing GPT-3.5-turbo with llama-2-70b on the Torch-AR dataset.

The authors construct two benchmarks for evaluating the model: RAPID (Recommend APIs based on Documentation), which encompasses four single-library API recommendation tasks and one multi-library task; LOCG (Library-Oriented Code Generation), that contains one single-library code generation task and one multi-library task. Experimental results on these benchmarks demonstrate the effectiveness of CAPIR in comparison to existing baselines like ADA-retrieve and CLEAR, with improvements in recall and precision of the recommended APIs, leading to better performance on library-oriented code generation tasks. Experiments with llama-2-70b show that its performance is lower than the originally proposed framework, probably because llama-2-70b is a base model without instruction tuning and thus has a weaker ability to follow instructions. Furthermore, the authors conduct a detailed qualitative analysis and experiments in realistic application scenarios, further validating that as long as the task descriptions are clear, CAPIR can accurately recommend the combination APIs for coarse-grained requirements and be effectively applied for practical applications

Chen et al.[51] point out that existing code completion approaches to API recommendation mainly rely on type information during compilation or heuristic rules to provide suggestions, resulting in a slow and often inaccurate process, and still require the developer to identify the correct alternative among those proposed. An effective API auto-completion framework helps programmers avoid spending much time on manually consulting documentation and manually entering the complete name of APIs, thus speeding up the writing time and reducing the risk f introducing common errors.

The authors therefore propose an API auto-completion method that leverages deep learning and transformer models to enhance the efficiency and accuracy of API recommendations by considering object types in the programming context.

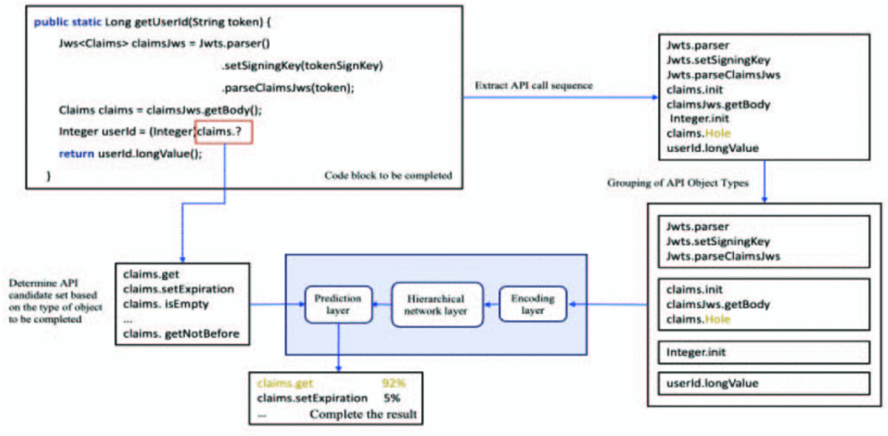


Figure 3.61: The overall framework for API completion.

The first step involves using static analysis techniques to extract the API call sequence from the programming context, replacing the position to be completed with the placeholder "Hole" -treated as a special API-, and grouping API calls according to object types. The grouped API call sequence is fed into the completion model. The encoding layer processes each API call using two word-embedding matrices, to encode the object types and APIs separately. These two word vectors are then spliced together, allowing the model to understand more effectively when API calls belong to the same class, as they would share the same first half of the encoding. The concatenated vector representations from the encoding layer go into the Hierarchical network layer, which further processes APIs and objects to obtain the vector representation of the entire programming context through the object state. This layer processes text information decomposing it into different levels and extracting features at each level. The API call is treated as a word, and the object state as a sentence, considering that the object state can be extracted from the API call sequence, and the programming context feature can be extracted from all object states. The programming context is divided into three parts to be processed: the API calls before the position to be completed, the position to be completed, and the API calls after the position to be completed. API calls before and after the position to be completed are grouped by object type and API calls of the same type are extracted. A transformer encodes the object state for each sequence of API calls, deriving an encoding representing the state information of the object. For the hole, a bidirectional transformer is used to understand both the context before and after the gap, allowing the model to predict the correct API considering the surrounding context. The object state sequence before the gap and the position to be filled are input into a forward transformer model to obtain the state before the gap and in the same way, the object state sequence after the gap is input into an inverse transformer model to obtain the state after the gap. Finally, these hidden states computed by first-layer transformers are used as input of a second-layer LSTM and combined to form a complete representation of the code block to be completed.

For the Prediction layer, the method state is commonly passed into a fully connected network with a dimension equivalent to the entire API, then the result is input into a softmax layer and the API with the highest probability represents the final completion result. However, since the dataset used in the study is very large, the authors devise another approach to make predictions. Indeed, since the type of API to be completed is known, the API call sequence associated with that type is determined. Then, the cosine similarity between the vector representation of each API in the candidate API set and the vector representation of the code block to be completed is computed and converted into a probability using the softmax function. This process is implemented by training a fully connected network with an output dimension equal to the dimension of word vectors, which allows reducing memory usage and improving model prediction speed. Next, each API's completion probability is sorted and the APIs with higher probability are recommended to users to fill in gaps in API calls.

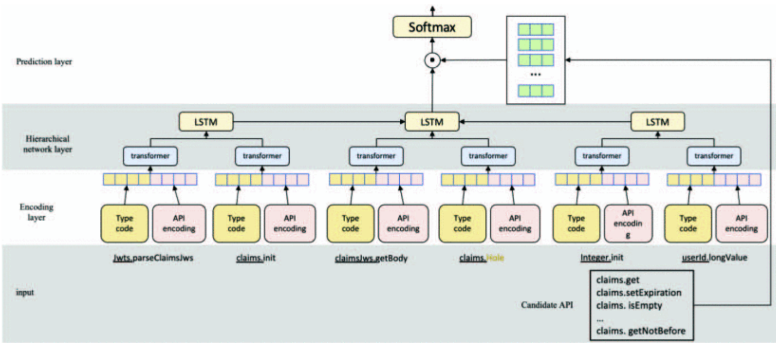


Figure 3.62: Structure diagram of API completion method.

The proposed solution is compared against baselines including N-Gram, LSTM, APIHelper, and Tang's approach. The N-Gram model takes the sequence of N-1 API calls around the gap position of the API to be predicted as input and calculates their probabilities with two models, a grammar model trained based on external code libraries, and a grammar model trained based on source code files under the current project and makes the API completion recommendation based on the joint probability distribution of the two models. APIHelper is based on the traditional LSTM model, but adds some new techniques, such as the splicing encoding method (which separates object types and API calls and then combines them into a single vector representation) and deterministic negative sampling during training (meaning that the model does not need to consider all possible APIs, but specifically samples object types related to the missing positions, leading to faster convergence and improved training efficiency). Tang's divides APIs into different object types using their semantic information and completes them based on the object types, taking into account the dependencies between APIs.

Experimental results show that the method proposed in the paper can recommend relevant APIs to developers based on the analysis of object types in the programming context, and outperforms the several compared API completion models from Top-1 to Top-10 accuracy, with a more significant advantage before Top-5.

Traditional query-based and code-based API recommendation techniques struggle with contextual understanding, scalability, and user input dependency. RQ5 reviews various pre-trained and transformer-based approaches, highlighting the relevant enhancements introduced.

query-based approaches

CLEAR[42] sets a foundation by leveraging a BERT-based model trained with contrastive learning to embed sentences from NL queries and Stack Overflow (SO) posts, learning the semantic similarity between sentence representations, regardless of their lexical information. After further filtering candidate posts, the re-ranking model, RoBERTa fine-tuned for classification with joint embedding training, computes the probability that a query and a given candidate post share the same APIs, ranks the selected candidate posts based on this score, and finally recommends the APIs from the top-ranked SO posts for the given query. Experimental results show that CLEAR significantly improves upon state-of-the-art approaches like BIKER , RACK, and DeepAPI, with a significant increase in mean average precision at both the method- and class-level.

PICASO[43] builds upon the CLEAR framework incorporating both code comments from GitHub and crowd-sourced knowledge from Stack Overflow to produce better API suggestions. In the first phase, PICASO identifies the most semantically relevant SO post to a given query (in the form of a code comment or annotation) using contrastive learning-based sentence embeddings and re-ranking. The second phase expands the query by adding the SO post's title and the APIs mentioned in its answers, and uses the extended queries to fine-tune CodeBERT to generate a sequence of API calls. Evaluations on a dataset combining GitHub annotations and SO posts demonstrate that PICASO significantly improves API recommendations, outperforming CodeBERT. This suggests that using additional context from SO helps refine vague queries by linking them to semantically relevant SO posts, and obtain more accurate API suggestions.

Huang et al. [44] propose a novel query clarification approach for API recommendation that leverages LLMs guided by a knowledge graph (KG). Traditional query clarification techniques face challenges such as out-of-vocabulary (OOV) failures and rigid template-driven question generation. To address OOV failures and generate more fluent, context-aware clarification questions, the authors use GPT-3.5. At the same time, the structured knowledge and entity relationships stored in the KG help filter out noise that may occur due to the randomness in LLM outputs and allow to transfer the optimal clarification path from KG to the LLM, increasing efficiency and control. The approach is structured in steps handled by separate LLM prompts: generating clarification questions, suggesting alternative options, extending the query based on prior interactions, and recommending APIs. Experiments show that the method significantly outperforms previous deep learning and KG-based baselines, while the integration of both LLM and KG compensates for their individual weaknesses, achieving high precision and robust API recommendations.

APICom[45] on the other hand, models the API recommendation task as an automatic completion task, where the correct API is generated based on a partial input provided by developers, including the query statement and known API prefixes. APICom uses random masking to create "prefix prompts" that simulate incomplete APIs by masking a certain number of words at the end of an API. To improve data quality and overcome the limitations of manually collected datasets, APICom performs data augmentation through an adversarial training method which, for each input prompt, generates adversarial examples. The generated adversarial examples are used alongside the original inputs as training data to fine-tune CodeT5 to predict the missing section of APIs and learn API completion patterns. Through evaluation on an extensive dataset of developer queries, APICom demonstrates superior performance compared to traditional recommendation models (e.g., BIKER, RACK), CLEAR and pre-trained models (e.g., CodeBERT, UniXcoder) on key metrics such as Exact Match@1, MRR, and MAP. The ablation studies also confirm the effectiveness of prompt learning and adversarial training in boosting APICom's overall performance.

In Mashup application development, where multiple open APIs are combined, finding suitable and high-quality APIs is challenging due to their large number and unstructured NL descriptions. Traditional methods often ignore the modeling of Mashup requirements and clusters, leading to poor accuracy and diversity in the recommended APIs. DLOAR[46] is a deep learning-based open API recommendation approach aimed at enhancing the precision and relevance of API recommendations in this context. DLOAR uses RoBERTa to generate high-quality word embeddings from Mashup descriptions and applies hierarchical clustering via HDBSCAN to cluster Mashups based on semantic similarity. The model then matches the requirements with the most relevant APIs at the cluster, Mashup, and API levels through a three-level similarity calculation. The system also provides additional information, such as API popularity and similar Mashup descriptions, to improve diversity and decision-making for developers. DLOAR commonly used methods (leveraging TF-IDF, LDA, Word2Vec, and Sentence-BERT) in precision, recall, and F1-measure.

While existing methods for Web API recommendations focus on either graphical or textual data, CME[47] integrates both knowledge graph embeddings and textual embeddings, ensuring a richer representation of mashup-API relationships. CME constructs a graph to represent relationships between mashups and APIs and uses Node2Vec to convert these entities into numerical vectors to be used for the downstream task, while BERT encodes mashup textual descriptions. Three fully connected layers within BERT are used to enhance feature stability and facilitate interaction between the text and knowledge graph domain, while two decoders are incorporated to decode the output of the BERT encoder into distinct semantic spaces (the graph embedding space and the API-specific space) and enhance knowledge exchange between them. During model training, an alternating training strategy optimizes both the Node2Vec and BERT models by aligning their representations until convergence, and facilitates the integration of textual and graphical data. The framework constructs three recommendation models (Classification, Retrieval, and Collaborative Filtering model)and evaluates them on the ProgrammableWeb dataset. The results show that CME, especially the Classification Mode, outperforms previous methods leveraging topic modeling, random walks, and deep learning, with higher precision, recall, and NDCG, particularly excelling in top-k API recommendations.

code-based approaches

PyBartRec[48] focuses on API recommendation for dynamic languages like Python, using a transformer-based model to address limitations in existing approaches, which rely on tools designed for static languages, that analyze the data flow information only and struggle with type inference. Instead, PyBartRec uses SPT-Code to extract semantic features from Python code snippets, enabling accurate API recommendations even when the APIs are not represented in the local data flow. The system also includes a post-processing type inference method that narrows down candidate types when initial type inference fails, enhancing the precision of API recommendations: PyBartRec processes the code context before the recommendation point, encodes the relevant features, and ranks candidate APIs based on these features. Evaluation on eight popular Python projects demonstrates that PyBartRec significantly outperforms existing solutions like PyART and Visual Studio Intellicode, achieving significantly higher top-k accuracy in intra-project API recommendations and MRR in cross-project recommendations.

PTM-APIRec[49] addresses key challenges faced by traditional API recommendation methods, such as the need for large training datasets and the inability to capture rich code semantics. It leverages general code pre-trained models to encode full code snippets, considering contextual information from the entire code rather than just API sequences, and uses separate general and API vocabularies, reducing training data requirements, improving adaptability to new API libraries, and ensuring only valid APIs are recommended. The system dynamically builds API vocabularies from source code and integrates static analysis to refine API predictions by considering object types and previous API calls and retrieving relevant API candidates. PTM-APIRec can adapt to different pre-trained and new API vocabularies without re-training the model from scratch. The evaluation results show that PTM-APIRec achieves high performance even when the size of the training data is small, thanks to the use of general pre-trained models. The approach outperforms code completion approaches (CodeGPT, CodeT5, UniXcoder, CodeBERT) and API recommendation approaches (n-gram model, APIHelper, APIRecX) on both top-k and MRR metrics, especially when incorporating models with the encoder component, like CodeBERT, CodeT5 and UniXcoder. Despite the longer inference time compared to simpler models and challenges with type inference for dynamic languages like Python, the results suggest that PTM-APIRec is particularly effective for static languages like Java, adaptable and suitable for evolving software environments and API libraries. CAPIR[50] focuses on API recommendation for library-based code and

low-resource scenarios, by breaking down coarse-grained user requirements into fine-grained subtasks and using an off-the-shelf embedding model to retrieve relevant APIs corresponding to each subtask. Traditional API recommendation methods often struggle with coarse-grained requirements that demand multiple fine-grained APIs. CAPIR addresses these challenges by using GPT-3.5-turbo and ada-embedding-002 in a three-step process: (1) it uses an LLM-based Decomposer to break down coarse-grained tasks into fine-grained subtasks using few-shot examples created by a Summarizer, which generates code summarizations and functional descriptions from API documentation, (2) retrieves relevant APIs for each subtask from documentation using an embedding-based Retriever, and (3) ranks the APIs using an LLM-based Reranker to ensure accuracy. CAPIR outperforms existing methods like ADA-retrieve and CLEAR, achieving higher precision and recall in API recommendation tasks across multiple benchmarks, including RAPID and LOCG. CAPIR's design ensures better alignment between user requirements and recommended APIs, improving library-oriented code generation performance in low-resource scenarios. *Chen et al.*[51] propose an API auto-completion method

that leverages deep learning and transformer models to enhance the efficiency and accuracy of API recommendations by considering object types in the programming context. Traditional methods often rely on type information or heuristic rules, which are slow and prone to errors. The new method extracts API call sequences from the programming context and groups them based on object types. A placeholder "hole" is used for the missing API, and the sequences are fed into the model. The method uses a hierarchical network, starting with an encoding layer that processes object types and API calls separately before concatenating them. Transformers then encode the object states, allowing the model to predict the missing API by analyzing the context before and after the "hole." A bidirectional transformer helps capture dependencies on both sides of the missing API, and a second-layer LSTM refines this to create a full representation of the code block. The approach reduces developers' reliance on manual documentation consultation, minimizes common coding errors, and speeds up code writing. Experimental results show that this method outperforms existing models, with significant improvements in Top-1 to Top-5 API completion accuracy. In conclusion, advancements in

Pre-Trained and Large Language Models mark a significant step forward in the automation of API recommendation systems by addressing the main shortcomings of previous approaches, such as handling contextual understanding, semantic ambiguity, the huge number of available APIs, the amount of training data required and the lack of diversity in recommendation outputs. By harnessing the power of LLMs, developers can benefit from enhanced quality of results, coding efficiency, and reduced cognitive load, with more scalable, adaptable, and accurate solutions.

Chapter 4

ChatGPT Experiment for Goal to API Mapping

This chapter presents a practical experiment on the application of LLMs in the field of requirements engineering.

4.1 Methodology

4.1.1 Approach

The experiment follows the approach outlined in [92].

The proposed framework is based on the Goal-Oriented Requirements Engineering (GORE) technique for eliciting, analyzing, and managing requirements, by modeling them as goals and sub-goals.

The approach comprises four main steps. The initial step involves analyzing the available documentation of a software project to extract high-level goals tailored to a specific stakeholder involved in the project, typically the software's end user. The following step involves breaking the high-level goals into low-level goals, potentially arranged in a hierarchical structure. The third step involves reading the JSON representation of the API and providing the list of available endpoints along with their information, including the textual descriptions, the available verbs on the endpoint, the type and name of parameters, the output type, and the provided result. The final step encompasses the mapping of the identified low-level goals to the software's existing API endpoints, which are retrieved from the software's API documentation provided as a Swagger file, or equivalent format. The final output is a set of API calls to one or more of the available endpoints, that can implement the low-level goals. If a given goal is not supported, a warning message should be returned, indicating that the mapping cannot be established using the current set of endpoints, which allows for further adjustment of the goal itself or the API. For each main step, additional prompts have actually been used when carrying out the experiment in order to revise and check the intermediate and final results.

4.1.2 Software project selection

The software projects for the experiment are selected from the EvoMaster Benchmark (EMB), a set of web/enterprise applications for scientific research in Software Engineering [93] [94]. The selection is based on an initial list comprising 16 projects and follows the criteria listed below.

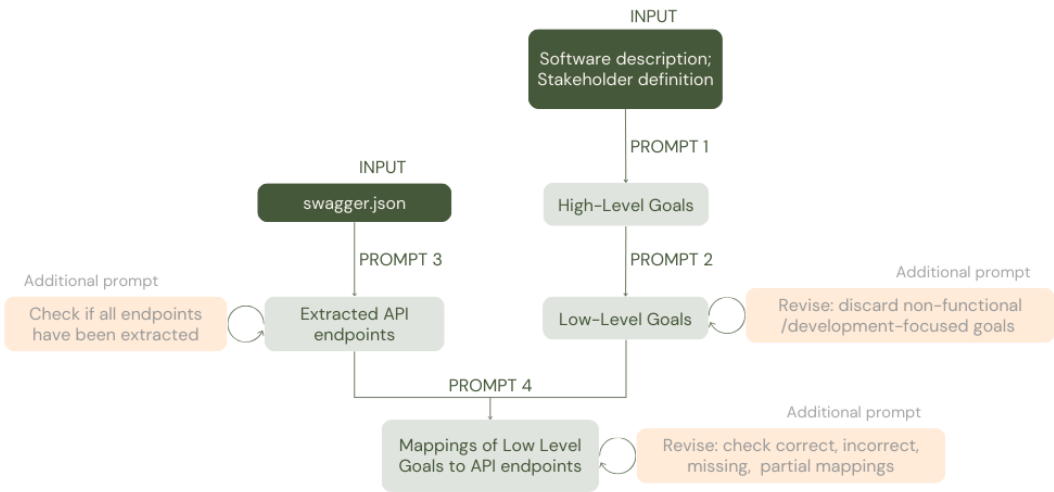


Figure 4.1: Process followed for the experiment.

Project selection criteria
The project is real or a demo and has a Swagger API description with textual documentation
APIs are documented or at least partly documented
The number of available endpoints is at least 7 and below 30

Table 4.1: Criteria for project selection

4.2 Results

The broader vision discussed in [92] is to realize an architecture that leverages multiple LLM-based agents operating autonomously to process a software description or its existing documentation and generate API calls that fulfill the software's functional requirements. This is part of an iterative process in which the LLM-based agents act in a loop to progressively refine the solution.

An initial experiment on the proposed approach consists of simulating the tasks that will be carried out by the individual agents through a series of conversational interactions with GPT-4 via the online ChatGPT interface.

As a case study, the detailed results on one of the selected software projects are presented, followed by the overall results for the four projects considered in 4.2.

4.2.1 Experiment for Genome Nexus

PROMPT 1: Generation of High-Level Goals

```
*Context:* You are assisting in the goal elicitation process for [Genome Nexus
is a comprehensive one-stop resource for fast, automated, and high-throughput
annotation and interpretation of genetic variants in cancer.  Genome Nexus
integrates information from a variety of existing resources, including databases
that convert DNA changes to protein changes, predict the functional effects of
protein mutations, and contain information about mutation frequencies, gene
function, variant effects, and clinical actionability]

*Stakeholder Description:* [Researchers]

*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
```

It can be seen from the results that some of the generated high-level goals pertain to non-functional aspects. These goals are: *2. High Throughput and Fast Processing, 4. User-Friendly Interface, 6. Real-Time Data Updates, 8. Robust Data Security and Privacy, 10. Scalable Infrastructure* . These goals should be identified and discarded instead of being considered for the alignment of API calls. Only in a few cases, throughout the experiment, it can be seen that, when splitting these non-functional high-level goals into low-level goals, some of the generated low-level goals might indeed be considered functional. Examples of this are the low-level goals generated for high-level goal *2. High Throughput and Fast Processing* (as seen below in the output to prompt 2, these are: *2.1 Batch Process Multiple Samples, 2.2 Monitor Processing Status*). Therefore, one could evaluate whether to include the non-functional high-level goals as input to prompt 2, and then revise and discard non-functional low-level goals output from prompt 2. The downside of this approach is the generation of many more low-level goals, and, most of all, the fact that the vast majority of the generated low-level goals will still be classified as non-functional.

1. Accurate and Comprehensive Variant Annotation - **Description:** Ensure the software provides precise and exhaustive annotation of genetic variants. - **Motivation:** Accurate annotations are crucial for researchers to reliably interpret genetic data and understand the implications of specific mutations in cancer research.

2. High Throughput and Fast Processing - **Description:** Enable the software to handle large volumes of genetic data swiftly and efficiently. - **Motivation:** Researchers often work with extensive datasets and need rapid processing to accelerate their research timelines and productivity.

3. Integration with Multiple Databases - **Description:** Facilitate seamless integration with a variety of genetic and clinical databases. - **Motivation:** Integrating diverse data sources ensures researchers have access to a comprehensive range of information, enhancing the depth and quality of their analyses.

4. User-Friendly Interface - **Description:** Develop an intuitive and easy-to-navigate interface for the software. - **Motivation:** A user-friendly interface reduces the learning curve and allows researchers to focus on their core research activities rather than struggling with the software.

5. Customizable Annotation Pipelines - **Description:** Provide flexible and customizable annotation pipelines to cater to specific research needs. - **Motivation:** Different research projects may require unique workflows; customizable pipelines enable researchers to tailor the software to their specific study requirements.

6. Real-Time Data Updates - **Description:** Ensure the software regularly updates its data sources to provide the most current information. - **Motivation:** Up-to-date data is essential for researchers to make informed decisions based on the latest findings and developments in the field.

7. Detailed and Accessible Reporting - **Description:** Offer detailed and easily accessible reports on annotated variants. - **Motivation:** Comprehensive reports help researchers quickly interpret their findings and share them with the broader scientific community or clinical collaborators.

8. Robust Data Security and Privacy - **Description:** Implement stringent security measures to protect sensitive genetic data. - **Motivation:** Ensuring data security and privacy is critical to maintaining the confidentiality of research subjects and compliance with ethical standards.

9. Collaborative Features - **Description:** Include features that support collaboration among research teams. - **Motivation:** Facilitating collaboration enables researchers to share insights and leverage collective expertise, advancing their projects more effectively.

10. Scalable Infrastructure - **Description:** Design the software to be scalable to accommodate growing datasets and user demands. - **Motivation:** Scalability ensures the software remains effective and efficient as research projects expand and evolve, preventing performance bottlenecks.

PROMPT 2: Generation of Low-Level Goals

```
**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 project.
The high-level goals of the software are the following: [RESULTS OF PROMPT P1]

*Task:* Based on your understanding of the typical tasks that compose the sequence
of high-level goals, provide if possible a decomposition of goals into sub-goals.
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 goals for the
implementation of the software.
```

The results across the various steps of the process show some inconsistency and variability among ChatGPT's outputs, starting from the generation of low-level goals. Since non-functional high-level goals (hlG) have been included as input for prompt 2, non-functional low-level goals (llG) have been generated, despite the instruction *"Each low-level goal should theoretically correspond to a single action of the actor with the software"*. Moreover, some goals appear to be more development-focused rather than directly aimed at the end-user actions (i.e. researchers), despite the instruction *"Please give goals from the perspective of the user using the application, not development goals"*.

To reason on these aspects and identify mistakes, other prompts were used to ask ChatGPT to determine if any of the generated goals covered non-functional aspects or was given from a development perspective.

In the context of the broader vision with a multi-agent architecture, a self-critique mechanism in charge of verifying goals could be used to refine the results and better identify and discard non-functional and development-focused goals.

1. **Accurate and Comprehensive Variant Annotation**
   (a) Upload Genetic Data Files - Description: Easily upload genetic data files in various formats. - Motivation: Researchers need a simple method to input their genetic data for analysis and annotation.
   (b) Select Annotation Parameters - Description: Choose specific annotation parameters and criteria. - Motivation: Researchers require the ability to tailor the annotation process to their specific research needs and focus areas.
   (c) View Detailed Variant Information - Description: Access comprehensive details on each annotated variant, including DNA changes, protein changes, and predicted effects. -

Motivation: Detailed variant information is essential for accurate interpretation and further research analysis.

2. **High Throughput and Fast Processing**

   (a) Batch Process Multiple Samples - Description: Submit multiple genetic samples for annotation in a single batch. - Motivation: Handling large datasets simultaneously saves time and increases efficiency for researchers working with extensive genetic data.

   (b) Monitor Processing Status - Description: Track the status and progress of data processing in real-time. - Motivation: Researchers need to monitor their data processing to manage their time effectively and ensure timely results.

3. **Integration with Multiple Databases**

   (a) Select Databases for Integration - Description: Choose which external databases to integrate with for annotation purposes. - Motivation: Researchers may prefer specific databases that best suit their research requirements and enhance their data's comprehensiveness.

   (b) Update Database Connections - Description: Regularly update and manage connections to external databases. - Motivation: Keeping database connections current ensures researchers have access to the latest and most relevant data.

4. **User-Friendly Interface**

   (a) Customize Dashboard Layout - Description: Personalize the dashboard layout to highlight frequently used tools and features. - Motivation: Customizable dashboards improve usability and help researchers quickly access the functions they need.

   (b) Access Help and Tutorials - Description: Easily find and use help documentation and tutorials. - Motivation: Ready access to support resources helps researchers quickly overcome any challenges or questions about using the software.

5. **Customizable Annotation Pipelines**

   (a) Create Custom Pipelines - Description: Design and save custom annotation pipelines for different research projects. - Motivation: Custom pipelines allow researchers to streamline their workflows and apply consistent analysis methods across similar projects.

   (b) Modify Existing Pipelines - Description: Adjust parameters and settings of existing annotation pipelines as needed. - Motivation: Flexibility in modifying pipelines ensures researchers can adapt to new findings or project requirements without starting from scratch.

6. **Real-Time Data Updates**

   (a) Schedule Automatic Data Updates - Description: Set up automatic schedules for data source updates. - Motivation: Automated updates ensure researchers always work with the most current data without manual intervention.

   (b) Receive Update Notifications - Description: Get notified when new data updates are available and applied. - Motivation: Notifications keep researchers informed about the freshness of their data and any changes that might impact their analysis.

7. **Detailed and Accessible Reporting**

   (a) Generate Comprehensive Reports - Description: Produce detailed reports summarizing annotated variant data. - Motivation: Comprehensive reports help researchers document their findings and share results with collaborators or for publication.
   (b) Export Reports in Multiple Formats - Description: Export reports in various formats (e.g., PDF, Excel). - Motivation: Flexible export options enable researchers to use and share reports in the most convenient format for their needs.

8. **Robust Data Security and Privacy**
   (a) Set Data Access Permissions - Description: Define and manage user access levels and permissions for data. - Motivation: Controlled access ensures sensitive genetic data is protected and only accessible to authorized personnel.
   (b) Encrypt Stored Data - Description: Automatically encrypt all stored genetic data. - Motivation: Encryption is crucial for maintaining the confidentiality and integrity of sensitive research data.

9. **Collaborative Features**
   (a) Share Data with Team Members - Description: Easily share annotated data and reports with team members. - Motivation: Collaboration tools facilitate teamwork and knowledge sharing, driving collective research efforts forward.
   (b) Collaborate on Annotation Pipelines - Description: Allow multiple researchers to work on and refine the same annotation pipeline. - Motivation: Collaborative pipeline development ensures diverse expertise is applied, improving the overall quality and robustness of the analysis.

10. **Scalable Infrastructure**
   (a) Adjust Processing Power - Description: Scale processing power up or down based on data volume and analysis requirements. - Motivation: Scalable processing capabilities ensure efficient handling of varying data loads, maintaining performance and reducing costs.
   (b) Monitor System Performance - Description: Continuously monitor the system's performance and resource usage. - Motivation: Performance monitoring helps researchers identify and address any bottlenecks or issues, ensuring smooth operation and efficient data processing.

In the end, fifteen low-level goals are kept for the subsequent stages of the experiment out of all the twenty-one originally generated shown above. In particular, the discarded goals are the following: 3.2 Update Database Connections, 6.1 Schedule Automatic Data Updates, 8.1 Set Data Access Permissions, 8.2 Encrypt Stored Data, 10.1 Adjust Processing Power, and 10.2 Monitor System Performance.

PROMPT 3: List of available endpoints

```
**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] The swagger file documenting the APIs of the software is attached.

*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.

Step 3 also shows some limitations: ChatGPT struggles to extract all paths from the documentation, which leads to the omission of some of the documented endpoints in the results. This partial examination could be attributed to the size of the analyzed file, which makes it challenging for the LLM to retrieve all items at once. When asked to investigate further, ChatGPT was able to verify and capture all remaining paths. Refining the prompt with additional instructions beside *"please list all the endpoints"*, could help improve the initial output of this stage.

PROMPT 4: Mapping between API endpoints and Low-Level Goals

```
**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:[RESULTS OF
PROMPT 2 DISCARDING NON-FUNCTIONAL LOW-LEVEL GOALS]
The endpoints of the software are the following:[RESULT OF PROMPT 3]

*Task:* please map each goal to a sequence of calls to the given endpoints. If
a goal is not mappable to an endpoint, please write that the goal cannot be
enforced by using the current set of endpoints.
```

In the fourth step, it can be seen that ChatGPT was able to only map four of the low-level functional goals to the available API endpoints: 1.2 Select Annotation Parameters, 1.3 View Detailed Variant Information, 2.1 Batch Process Multiple Samples, 7.1 Generate Comprehensive Reports. In the example with the Genome Nexus project, the experiment was conducted in multiple chats, and additional prompts for clarification or further reasoning on the results were used. The outputs showed some variability, as using the same input prompts across different sessions can deliver different results and reasoning approaches: across the various sessions and for the same goal, the results do indeed present differences. For example, mappings are identified in some cases while not in others; some answers omit one or two endpoints compared to others.

Despite these observations though, the general direction is common to all chats. More aligned and comprehensive results are delivered when each goal is assessed individually, thus repeating prompt 4, and when the question to chatGPT includes more explicitly the request to map the goal to a comprehensive sequence of calls to the given endpoints.

The results of prompt 4 are in 4.2 below. Again, the results have been achieved through several interactions with the ChatGPT interface.

Low-Level Goal	Mapped Endpoints	Comment
1.2 Select Annotation Parameters	/annotation, /annotation/{variant}, /annotation/dbsnp/, /annotation/genomic, /annotation/dbsnp/{variantId}, /annotation/genomic/{genomicLocation}	Use query parameters like *fields* to select specific annotation fields for customization.
1.3 View Detailed Variant Information	/annotation/genomic/genomicLocation, /annotation/dbsnp/{variantId}, /annotation/{variant}, /annotation, /annotation/dbsnp/, /annotation/genomic	Provide detailed information for each annotated variant.
2.1 Batch Process Multiple Samples	/annotation, /annotation/genomic, /annotation/dbsnp/	Allow submitting a list of genomic locations for annotation in a batch.
7.1 Generate Comprehensive Reports	/annotation, /annotation/dbsnp/, /annotation/genomic	Provide the raw data needed to compile a report.

Table 4.2: ChatGPT goal-to-api mapping experiment results on Genome Nexus

The following paragraphs provide a more in-depth insight into the process and results. In

most sessions, ChatGPT's response for **Goal 1.1 Upload Genetic Data** (description: "easily upload genetic data files in various formats") is that the goal is not directly mappable with existing endpoints, while in others the following mappings are established, including /annotation, /annotation/dbsnp, /annotation/genomic. In these cases, an additional prompt was used to ask ChatGPT to evaluate the correctness of the mappings, taking into consideration the documentation and the description of the goal. The final answer was that none of the identified endpoints is actually specifically designed to support file uploads or handle various file formats; instead, they are all primarily focused on retrieving annotation data, plus, they assume data in the form of a list of variants, dbSNP IDs or genomic locations rather than raw or formatted genetic data files such as MAF (Mutation Annotation Format) or VCF (Variant Call Format) files. These results could suggest the tendency to ignore parts of the input and (mis)interpret tasks based on surface-level similarity or assumptions rather than fully considering endpoints and goal specifics and checking the ones against the others to define potential mappings.

For **Goal 1.2 Select Annotation Parameters** (description: "choose specific annotation parameters and criteria") no mapping is established at first in any of the sessions. The reason reported across the various chats is that the existing endpoints lack the functionality to support user-defined customization of annotation parameters or criteria, providing only predefined outputs without options to dynamically adjust or select specific annotation types, filtering criteria, or output fields. Through an additional prompt or even by simply assessing the goal individually, the following endpoints are suggested as suitable: /annotation, /annotation/variant, /annotation/dbsnp/, /annotation/genomic, /annotation/dbsnp/variantId, /annotation/genomic/genomicLocation, with customization capabilities available through the query parameters isoformOverrideSource and fields. Parameter isoformOverrideSource is used to specify which set of canonical transcripts (mskcc or uniprot) should be used when annotating mutation data with Genome Nexus; parameter fields is used to specify which annotation fields to include in the annotation response (like annotation_summary, hotspots, mutation_assessor, nucleotide_context, oncokb).

Goal 1.3: View Detailed Variant Information (description: "view Detailed Variant

Information") is mapped to endpoints that retrieve VEP annotations. Specifically, endpoints /annotation/genomic/genomicLocation and /annotation/dbsnp/variantId are included in the results for all sessions: they directly align with the goal by providing a VEP annotation, which includes details such as DNA and protein changes and effects for variants located at a specified genomic location or based on a dbSNP identifier. After further interaction, ChatGPT ends up additionally including the following other mappings, in all chats: /annotation/variant, /annotation, /annotation/dbsnp/, /annotation/genomic. Indeed, according to their summary in the API documentation, they all allow retrieving variant annotation information, either by variant IDs, genomic locations, or dbSNP IDs, either individually or in bulk.

The mappings identified for **Goal 2.1: Batch Process Multiple Samples** (description: "submit multiple genetic samples for annotation in a single batch") are /annotation, /annotation/genomic, and /annotation/dbsnp/ (that was initially missing in the answers of some of the chats). These endpoints allow submitting a list of variants, genomic locations, or dbSNP IDs in one request to retrieve their annotation data; however, not all of them are included in the results in the first answer, but only after further asking.

For **Goal 5.2: Modify Existing Pipelines** (description: "researchers should be able to adjust parameters and settings of existing annotation pipelines as needed"), no mapping is found. The reasoning is that many endpoints (/annotation, /annotation/genomic, /annotation/dbsnp/, /annotation/dbsnp/variantId, /annotation/variant, /annotation/genomic/genomicLocation), allow specifying parameters isoformOverrideSource and fields, which offer minor adaptability of the output of the annotation process to researchers' needs on a per-query basis; however, these endpoints do not allow for modifications to the pipeline itself, they do not modify or save changes in settings and configurations of an existing pipeline, providing a mechanism for persisting these adjustments. Thus, a new API functionality for pipeline management would need to be implemented to support this goal. Finally, for **Goal 7.1: Generate Comprehensive Reports** (description:

"produce detailed reports summarizing annotated variant data"), no mapping is established at first; after an additional prompt, ChatGPT suggests endpoints that retrieve the data that would be needed to generate a report: /annotation, /annotation/dbsnp/, /annotation/genomic. The data provided by these endpoints can then be processed and formatted externally to create a comprehensive report and meet the goal's requirements: once retrieved, the data could be processed and aggregated as needed, while generating, formatting, and exporting the report itself being managed by client-side applications or external tools. Endpoints providing detailed annotations for individual variants or genomic locations, or those focused on retrieving genes, transcripts, or post-translational modification data are not taken into consideration. /ensembl/canonical-gene/entrez, /ensembl/canonical-gene/hgnc, /ensembl/canonical-transcript/hgnc, /pfam/domain, /ptm/experimental, /ensembl/transcript could be considered supplementary to the goal as they provide data that could enrich a variant report, however, if the goal strictly requires annotated variant data alone, they would be out of scope as they do not focus directly on variant-specific annotations. In conclusion, the alignment between low-level goals and API endpoints is not an

easy task to perform. The limited success in achieving mappings can be partly attributed to the available endpoints, which do not fully support many of the functionalities implied by the goals. In these regards, better results could likely be achieved with richer API documentation. Beside that, ChatGPT's responses show a certain variability and occasional inconsistency: the

LLM is able to identify potential mappings for some goals, but it sometimes makes assumptions based on surface-level observations rather than fully considering endpoints and low-level goals' descriptions. The overall results of the experiment confirm that the approach would benefit from a structured agent architecture, with iterative refinement and self-critique to better align with the requirements of each goal and achieve more accurate mappings with fewer misinterpretations; the usefulness of a multi-agent architecture with step-by-step analysis and refinement of intermediate and final results is clear.

	LLG name	LLG description	Mapping to endpoints
1.1	Create a New Product	Allow users to define a new product by specifying its name, description, and unique identifier	/products/productName
1.2	Add Features to a Product	Enable users to add features to an existing product, including details such as feature name, description, and constraints	/products/productName/ features/featureName
1.3	Define Feature Constraints	Allow users to specify activation constraints between features of a product	/products/productName/constraints/ requires, /products/productName/constraints/ excludes
2.1	Create a Product Configuration	Allow users to create a new product configuration by selecting and activating a set of features for a product	/products/productName/ configurations/configurationName
2.2	Modify an Existing Configuration	Enable users to modify an existing product configuration by adding or removing active features	/products/productName/ configurations/configurationName/ features/featureName
2.3	Validate Configuration Constraints	Provide functionality to validate a configuration against the defined feature constraints	-
3.1	Query Active Features of a Configuration	Allow users to query and retrieve the list of active features for a given product configuration in real-time	/products/productName/ configurations/configurationName /features
3.2	Filter Features by Criteria	Enable users to filter active features based on specific criteria such as user role or client type	-
5.1	Manage User Access Control	Enable users to define and manage access control policies for different features and configurations	-
6.2	Provide User Support Channels	Offer support channels such as forums, chat, or email to assist users with any issues or questions	-
10.1	Collect User Feedback	Provide mechanisms for users to submit feedback on their experience with the microservice	-

Table 4.3: Low-Level Goals (LLGs) for Features-Service

Name	Endpoints	HLGs	LLGs	LLGs after discarding non-functional and development goals	LLGs correctly mapped
Genome Nexus[95]	23	10	21	16	4
Features-Service[96]	18	10	22	11	6
Market[97]	13	10	21	18	9
Reservations API[98]	7	10	30	17	6

Table 4.6: Reports the selected software projects along with: number of available API endpoints; high-level goals and corresponding low-level goals; number and percentage of low-level goals for which a mapping was found

The experiment reveals critical insights into ChatGPT's performance and effectiveness in mapping software goals to API endpoints across the four projects. It can be deduced that performance is influenced by several factors. The quality of the software description plays an important role in generating relevant high-level goals, as incomplete or ambiguous descriptions often lead to vague or redundant goals. Additionally, there is a tendency to include non-functional or development-focused goals, which highlights the need for careful prompt engineering to prioritize end-user objectives.

	LLG name	LLG description	Mapping to endpoints
1.1	Search for Products	Users can enter keywords to search for specific products	-
1.2	Filter Products by Category	Users can filter products based on predefined categories	-
1.3	Sort Products by Price, Rating, and Popularity	Users can sort products by various criteria such as price, customer rating, and popularity	-
2.1	Add Products to Cart	Users can add selected products to their shopping cart	/customer/cart
2.2	Remove Products from Cart	Users can remove unwanted products from their shopping cart	/customer/cart
2.3	Update Product Quantity in Cart	Users can adjust the quantity of each product in their shopping cart	/customer/cart
3.1	Register a New Account	Users can create a new account by providing necessary personal information	/register
3.2	Login to Existing Account	Users can log in to their existing account using their credentials	-
3.3	Reset Forgotten Password	Users can reset their password if they forget it	-
4.1	Proceed to Checkout	Users can initiate the checkout process from their shopping cart	/customer/cart/pay
4.2	Select Shipping Options	Users can choose their preferred shipping method during checkout	/customer/cart/delivery
4.3	Track Order Status	Users can view the current status of their orders from their account	/customer/orders
5.1	View Past Orders	Users can access a list of their previous orders from their account	-
5.2	Repeat a Past Order	Users can quickly reorder items from their past orders	-
6.1	Contact Customer Support	Users can contact customer support through various channels such as chat, email, or phone	-
6.2	Access Help Articles and FAQs	Users can browse help articles and frequently asked questions for self-service support	-
9.1	Receive Product Recommendations	Users receive personalized product recommendations based on their browsing and purchase history	-
10.1	Choose Payment Method	Users can select from various secure payment methods during checkout	/customer/cart/pay

Table 4.4: Low-Level Goals (LLGs) for Market

One is the type of software project: user-centric applications with a simpler structure and more direct end-user focus, like the Reservation System and Market, generally yield better results in contrast to more complex systems.

Also, while vague or overly technical descriptions may lead to confusing and ambiguous goals, necessitating manual refinement, it is likely that a richer description in the case of more articulated projects, like Genome Nexus and the Features System, enables ChatGPT to generate more varied, aligned and actionable goals, because it provides clearer context, functionalities, and nuanced stakeholder needs. On the other hand, in simpler contexts, such as the Basic Reservation System, even a less detailed description sufficed for generating reasonably aligned goals. Indeed, simpler systems often involve more universal, straightforward functionalities (e.g., booking, notifications, or user profiles) that ChatGPT can more easily infer from even minimal input.

Another aspect to consider is that the goals generated for more articulated scenarios like the Genome Nexus project often extended beyond the provided endpoints, and therefore remained unmapped due to missing API functionalities, rather than ChatGPT's inability to perform the task. In these cases of course a richer API documentation would maximize mapping capabilities.

On the other hand, some goals might overlap, referring to the same base functionality. Examples of this can be seen for the goals related to reservation submission in the Reservation

	LLG name	LLG description	Mapping to endpoints
1.2	Simple Reservation Form	The reservation form should be straightforward and easy to complete	/api/reservation
2.2	Auto-Fill Forms	Forms should have auto-fill options based on previous user data	/api/reservation/username
3.1	Live Calendar View	Show a live calendar view with up-to-date availability	-
3.3	Availability Alerts	Provide alerts for changes in availability for desired slots	-
4.2	Multiple Payment Options	Offer multiple payment options (credit card, PayPal, etc.)	/customer/cart
4.3	Transaction Confirmation	Provide immediate transaction confirmation and receipt	/customer/cart
6.1	Instant Email Confirmation	Send an instant email confirmation upon reservation completion	/register
6.2	SMS Reminders	Send SMS reminders prior to the reservation time	-
6.3	Notification Preferences	Allow users to set their notification preferences (email, SMS, app alerts)	-
7.1	Easy Cancellation Option	Provide a straightforward option to cancel reservations	/api/reservation/reservationUuid
7.2	Reschedule Functionality	Allow users to reschedule reservations without hassle	/api/reservation
8.1	Save User Preferences	Enable users to save their preferences in their profiles	/customer/orders
8.2	View Booking History	Allow users to view their past bookings and details	/api/reservation/username
8.3	Profile Management	Provide options for users to update their profile information easily	-
10.1	In-App Help Center	Include an in-app help center with FAQs and tutorials	-
10.2	Live Chat Support	Provide live chat support for real-time assistance	-
10.3	User Feedback System	Allow users to submit feedback and report issues directly through the system	-

Table 4.5: Low-Level Goals (LLGs) for Basic Reservation System

System (*2.1 Quick Reservation Button* and *1.2 Simple Reservation Form*), but also for transaction confirmation (*4.3 Transaction Confirmation* and *6.1 Instant Email Confirmation*). This can create ambiguity and emphasizes the need for more goal refinement before mapping. The overlap might more likely happen across goals of simpler projects or when software description is limited, whereas with more detailed descriptions and projects it is a less likely scenario.

A significant tendency observed is the inclusion of non-functional or development-focused goals, particularly in more complex projects (e.g., Genome Nexus and Features-Service). This suggests that ChatGPT may struggle to consistently distinguish user-facing and technical goals, and would benefit from more explicit guidance in the prompts or refinement of intermediate results. Also, another characteristic that can be observed in the generated goals (both high- and low-level, but especially high-level ones), is that they are often accompanied by adjectives that qualify them beyond the mere functionality they are expressing. As examples, we can look at the high-level goals for the Market application: *1. User-Friendly Product Browsing, 2. Efficient Shopping Cart Management, 3. Simple User Registration and Login, 4. Convenient Order Placement and Tracking.*

Concerning the retrieval of API endpoints from Swagger documentation, the process is efficient and only in the case of Genome Nexus, which has the highest number of endpoints -though it is not extensive-, ChatGPT did not retrieve them all at first, but this can likely be solved by refining instructions in the prompt.

4.3 Prompt Engineering for Improvement

Despite the challenges highlighted for the experiment in Chapter 4, ChatGPT shows reasonable potential in automating the stages of the goal-to-endpoint mapping process, reducing the manual workload required for goal elicitation and endpoint association. However, the need for human oversight and intervention seems to remain crucial, especially considering that the software scenarios can get more complex than those used for the experiment.

Below are suggestions to improve the prompts involved in the experiment:

- **Provide definitions of functional and non-functional goals**, along with relevant examples, to enrich either Prompt 1 or Prompt 2. If this additional information is added to Prompt 1, it may be advisable to remove the instruction to consider "Quality and Performance Expectations" when generating high-level goals. Including this instruction might inadvertently steer the output toward non-functional requirements. By focusing on what the software should do (instead of how it should perform), the prompt can better guide the model to generate user-oriented, action-driven goals.

- **Explicitly specify the desired number of high-level and low-level goals** the model is expected to produce. For example, applying Prompt 2 individually to each high-level goal may generate more valuable low-level goals, but when done without specific limitations, it tends to generate a significantly larger number of goals. When all high-level goals are included in Prompt 2, the model typically produces 2-3 low-level goals per high-level goal, while applying the prompt to each high-level goal individually can result in as many as 10 low-level goals per high-level goal. While this approach provides more comprehensive coverage, the increased volume of information might overwhelm the model, potentially causing undesirable behavior.

- **Make the language more concise and clear** where possible, eliminating unnecessary complexity and potential ambiguities. This can include simplifying lengthy parts or removing words or sentences that can potentially create ambiguities in results. This suggestion applies to all prompts. For instance, the software description in Prompt 1 could be refined, especially when it touches on more technical or complex aspects. In Prompt 2, removing the word "theoretically" from the instruction "Each low-level goal should theoretically correspond to a single action of the actor with the software" may help reduce the likelihood of overlapping goals and better align the results with user actions in the system. Additionally, abstract or vague terms or phrases should be rephrased or removed. As an example, the sentence "Based on your understanding of the typical tasks that compose the sequence of high-level goal" in Prompt 2 could be rewritten for greater precision and clarity. In prompt 3, potential ambiguities might lie in the term "parameters", as in Swagger, parameters can appear in different contexts, such as query, path, header, or body. To resolve this, the prompt should explicitly instruct the model to include all parameters required to call the endpoint, specifying their type (e.g., query, path, header, body). These adjustments will ensure the output is complete, precise, and aligned with the user's expectations.

- **Using prompting techniques** such as n-shot prompting, role prompting, or step-by-step reasoning.

  - In n-shot prompting, the model is provided with illustrative examples of input-output pairs in order to better learn the desired task and produce more relevant results. For instance, in the context of Genome Nexus, a high-level goal might be: "Enable users to query genomic variants effectively". Corresponding low-level goals for this high-level objective could include: "Develop a search feature that supports querying by gene name, variant ID, and position", and "Implement filters to refine search results based on clinical relevance, mutation type, and frequency". In the case of prompt 4, these examples would represent mapping results.

– In role prompting, the model is instructed to assume a specific role or perspective to tailor its responses and align them with the context or task requirements. This technique could be particularly useful in Prompt 1 to improve the generation of high-level goals tailored to the specified stakeholder. For instance, the "Task" section of the prompt could instruct the model as follows: "Imagine you are a researcher using Genome Nexus. Consider your main priorities and needs when interacting with the software, and generate high-level goals from this perspective." This framing might help the model adopt a more stakeholder-specific approach, ensuring the goals are relevant and actionable.

– In step-by-step reasoning, the model is guided to break down tasks into steps, promoting more accurate and thoughtful responses. As an example, an improved version of Prompt 1 might read: "Break the task into steps: (1) Identify the stakeholder's main priorities and needs; (2) List high-level goals to address these priorities and needs; (3) Provide a one-sentence description and motivation for each goal." For prompt 3, the step-by-step instruction pattern might look like this: (1) Parse the provided Swagger file; (2) Identify all API endpoints; (3) For each endpoint, extract the following information: [list of attributes]; (4) list all the endpoints and their corresponding information. This approach can help the model produce more organized and meaningful outputs. For prompt 4, the step-by-step reasoning might take the following form: (1) For each goal, identify its requirements; (2) Compare these with the functionalities of each endpoint; (3) map each goal to a suitable endpoint or a sequence of the given endpoints"

These strategies can collectively help ensure that the prompts are more effective in guiding the model to produce outputs that are detailed, accurate, and aligned with the objectives of the experiment.

- In the step of extracting endpoint information from swagger documentation, it has been observed that with an increasing number of endpoints, ChatGPT might tend to include only a subset of endpoints in the first answer. This is especially true when the information is returned as a bullet point list, whereas when a table format for the output is used, it generally delivers complete results. Therefore, prompt 3 might ask to list or present the extracted information in a structured table format. This can also be downloaded and used as input to prompt 4 in the form of an attached file rather than listing all endpoints and corresponding information in plain text inside the prompt. In contrast with the current design of prompt 3, applying the step-by-step reasoning approach above, already generally returns a table rather than the textual list of endpoints.

- **Include instructions for handling ambiguities, and partial or indirect mappings** to prevent unnecessary speculation or misinterpretation of the task. For example, the prompt could specify: "If a goal can only be partially met with the available endpoints, state which aspects of the goal are addressed and which are not. Similarly, if a goal is only indirectly achievable with the available endpoints, provide a clear explanation of how the endpoints relate to the goal indirectly". Beside this additional instruction, another beneficial approach to use is assessing each goal individually in the prompt. This seems to provide more reasoned and complete mappings, enriched with clearer explanations.

- **Include additional prompts to refine results of main prompts**. For example, consider that even with the improvement strategies seen so far, the goals generated as output to prompts 1 and 2 might still be related to non-functional or development aspects. To address this, the experiment could benefit from additional prompts to classify the goals. These prompts could either evaluate each low-level goal individually or assess all goals collectively to establish whether they are functional or non-functional and development-oriented or end-user-focused. Based on this classification, the goals can then be excluded from the subsequent mapping phase, while possibly being annotated separately if providing any potentially helpful insight for the software under consideration. Prompt patterns such as *Question Refinement* and *Cognitive Verifier* have been suggested to possibly enhance requirement classification. *Cognitive Verifier* aims to improve LLM reasoning by breaking high-level questions into smaller, more manageable sub-questions, that provide answers which are then combined into the overall answer to the original question. *Question Refinement* actively engages the LLM in the prompt engineering process, asking it to suggest an improved or more refined version of the user's question to find the right way to pose it in order to get an accurate answer. Another example of additional prompt that could be integrated into the process is one aimed at addressing ambiguous goals and refining them for mapping to the API endpoints, following a Specification Disambiguation pattern [99].

Along with improved versions of the experiment's prompts and additional prompts to check on intermediate results, an AI agent architecture incorporating planning, action, observation, and reflection, with iterative feedback and refinement mechanisms, would likely bring significant enhancements to the elicitation and mapping process, while also making it more suitable for larger projects. While this Chapter focused on experimenting the Goal-to-API Mapping process via the online ChatGPT interface, the next Chapter describes the implementation of a sample code architecture leveraging Large Language Models to achieve the same objective, while also including an iterative mechanism for evaluation and improvement of intermediate outputs, the possibility to adopt different LLM providers and models and track usage data.

Chapter 5

LLM-Based Architecture for Goal to API Mapping

5.1 Overview

This Chapter describes the approach adopted for the development of an architecture leveraging Large Language Models (LLMs) to assist in the goal elicitation process and API mapping within a given software project context [100]. The architecture comprises distinct phases: generation of high-level goals (HLG), refinement into low-level goals (LLG), endpoints information extraction from API documentation, mapping of low-level goals to available endpoints. The design includes a feedback loop with a judge mechanism in charge of evaluating the outputs of each phase based on specified quality criteria (such as a completeness and correctness score) and of providing suggestions for subsequent improvement if needed.

Below are the key aspects of the architecture

- **LLM Abstraction:** The system abstracts interactions with various LLM providers (e.g., OpenAI [101], Google [102], Anthropic [103]) via a unified client instance (LLMClient). This abstraction simplifies switching between providers while keeping the core logic consistent. It also encapsulates provider-specific logic and token usage tracking.

- **Iterative Feedback and Retry Mechanism:** The design incorporates an iterative evaluation loop where the generated output is evaluated against predetermined criteria. If the output is judged not to meet the criteria, an improvement prompt is issued, and the process is repeated up to a maximum number of iterations. This aims to provide and internal mechansim for enhancing quality of the generated content.

- **Domain-Specific Actions:** The architecture is organized into four core phases. These include generating High-Level Goals using the project description as input; generating Low-Level Goals by refining high-level goals into actionable, user-centered tasks; extracting endpoints from a Swagger file and finally aligning them with the previously generated goals.

- **Utility Functions:** Supplementary functions handle tasks such as cleaning output, saving results to files, and flattening JSON responses into plain text when needed.

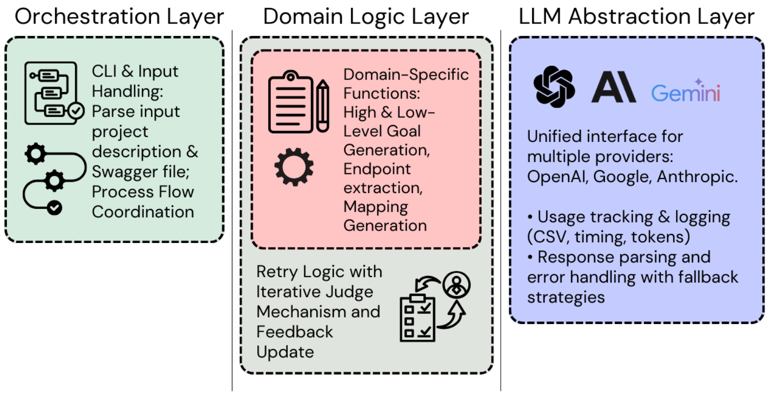


Figure 5.1: Conceptual Layers for the Goal-to-API architecture.

5.2 Detailed Analysis

This section aims to delve deeper into the details of how the architecture is actually implemented.

5.2.1 Domain-specific functions

The domain-specific functions are the functions representing the fundamental phases of the Goal-to-API mapping process. The whole process requires passing a .txt file containing the description of the software project to analyze and the API documentation file in .json format via command line. The prompts used in this context have been changed and simplified with respect to those leveraged for the ChatGPT experiment.

High-Level Goals Generation The function `generate_hlg` takes a project description (supplied via a text file) and executes a call to the LLM sending a prompt that asks for the elicitation of a list of high-level goals based on the project description extracted from the file. These goals are intended to capture the primary functionalities or services that an end-user is expected to require from the system.

Low-Level Goals Decomposition Next, the function `generate_llg` sends a prompt to the LLM passing the final high-level goals generated, and asks for refinement into low-level goals. This phase aims to produce more detailed, actionable sub-goals that describe individual functionalities and are therefore more suitable for mapping to API endpoints later in the process.

API Endpoints Extraction Using the function `extract_endpoints`, the system parses a the JSON file to extract the available API endpoints along with their descriptions. The resulting output is a structured, textual representation of the endpoints.

Mapping Goals to API Endpoints Finally, the function `match_goals_to_api` invokes the LLM passing low-level goals and extracted API endpoints to the prompt, and returns mappings between the two when possible. As an example, the code for `match_goals_to_api` is reported:

```
def match_goals_to_api(low_level_goals, extracted_endpoints, feedback=""):
prompt = f"""You are mapping user goals to API endpoints.\n
            Goals:
            {low_level_goals}
            \nEndpoints:
```

```
            {extracted_endpoints}
            Map each goal to the most appropriate endpoint.
            {f'\nAlso consider: {feedback}' if feedback else ''}"""

    return llm.call_llm(prompt, response_format=Result, model=llm.model)
```

5.2.2 Feedback and Evaluation

The function `action_with_retry` provides a generic mechanism for executing an action with a retry loop, essentially wrapping around the domain-specific actions (`generate_hlg`, `generate_llg`, `match_goals_to_api`). `action_with_retry` calls a given domain-specific function providing the appropriate input data, then extracts the textual output, and passes it to another function, `judge_output_iterative`. This function leverages an LLM acting as a judge [104][105] for evaluating the quality of the generated content and returns the original output if positively evaluated on the first attempt, or a suggestion for improvement otherwise. `action_with_retry` then updates the input data appending the newly generated suggestion to the `feedback` parameter (using a helper function `update_feedback`), and retries the action with the updated input data until the output is accepted or the maximum number of attempts is reached.

```
def action_with_retry(action_name, action_function, input_data, input_modifier,
judge_context="", max_attempts=3):
output = None
for attempt in range(max_attempts):
    try:
        result = action_function(**input_data)
        output = result.output
        judged_output = judge_output_iterative(action_name, output,
        additional_context=judge_context)

        if judged_output.strip() == output.strip():
            print(f"[{action_name}] Accepted at attempt {attempt + 1}")
            return output
        else:
            input_data = input_modifier(input_data, judged_output)
    except Exception as e:
        print(f"[{action_name}] Failed on attempt {attempt + 1}: {e}")

if output is None:
    raise RuntimeError(f"All attempts for {action_name}
    failed with no valid output.")
return output
```

In this implementation, the judge evaluates the output of the domain specific actions according to predefined metrics —in this case, assigning scores for attributes such as how well goals reflect end-user expected functionalities, or how exhaustive are the mappings—. These evaluation criteria differ based on the domain-specific function under analysis. If the scores do not meet a threshold (for example, a score of at least 8/10), the judge provides a suggestion for improvement.

```
def judge_output_iterative(action_name, output, additional_context="",
max_iterations=3):
```

```
    for iteration in range(max_iterations):
        evaluation_info = get_evaluation_criteria(action_name)
        prompt = (
            f"Judge result of action '{action_name}'.\n"
            f"RESULT TO EVALUATE:\n{output}\n\n"
            f"{f'Consider the following context: {additional_context}\n'
            if additional_context else ''}"
            f"Here is the information you should provide:\n{evaluation_info}"
            f"Format your response EXACTLY as a JSON object. Below is an example:\n"
            f"{{\n"
            f'    "Evaluation1": 8,\n'
            f'    "Motivation1": "Explanation for score given for Evaluation1",\n'
            f'    "Evaluation2": 8,\n'
            f'    "Motivation2": "Explanation for score given for Evaluation2",\n'
            f'    "OK": True,\n'
            f'    "suggestion": "Optional suggestion provided
            if any score is below 7, for improving the result
            of action {action_name} "\n'
            f"}}"
        )
        try:
            judge_result = llm.call_llm(prompt, response_format=JudgeResult,
            model=llm.model)
        except Exception as e:
            print(f"[Judge Iteration {iteration + 1}] Parsing failed: {e}")
            continue

            ...

        if judge_result.OK == True:
            return output
        elif judge_result.suggestion:
                return judge_result.suggestion
```

This design centralizes the logic for handling feedback and retry in an iterative process. Then, the `main` function performs the Goal-to-API flow simply by sequentially calling `action_with_retry` on the domain-specific functions.

5.2.3 LLM Abstraction

The `LLMClient` class provides a unified interface for interacting with multiple LLM providers (OpenAI, Google and Anthropic), abstracting away the differences between each provider's API and thus allowing the rest of the application to remain agnostic of the underlying LLM service in use. With this approach, the system remains flexible and configurable, facilitating experimentation and leaving room for comparison among multiple providers.

During initialization, the appropriate client is instantiated based on the provided configuration in the `.env` file, which specifies provider, model and API key.

The actual calls to the LLM are handled by the `call_llm` method, which serves as a generic wrapper that sends a prompt to the LLM, and returns the response. One key aspect of `call_llm`

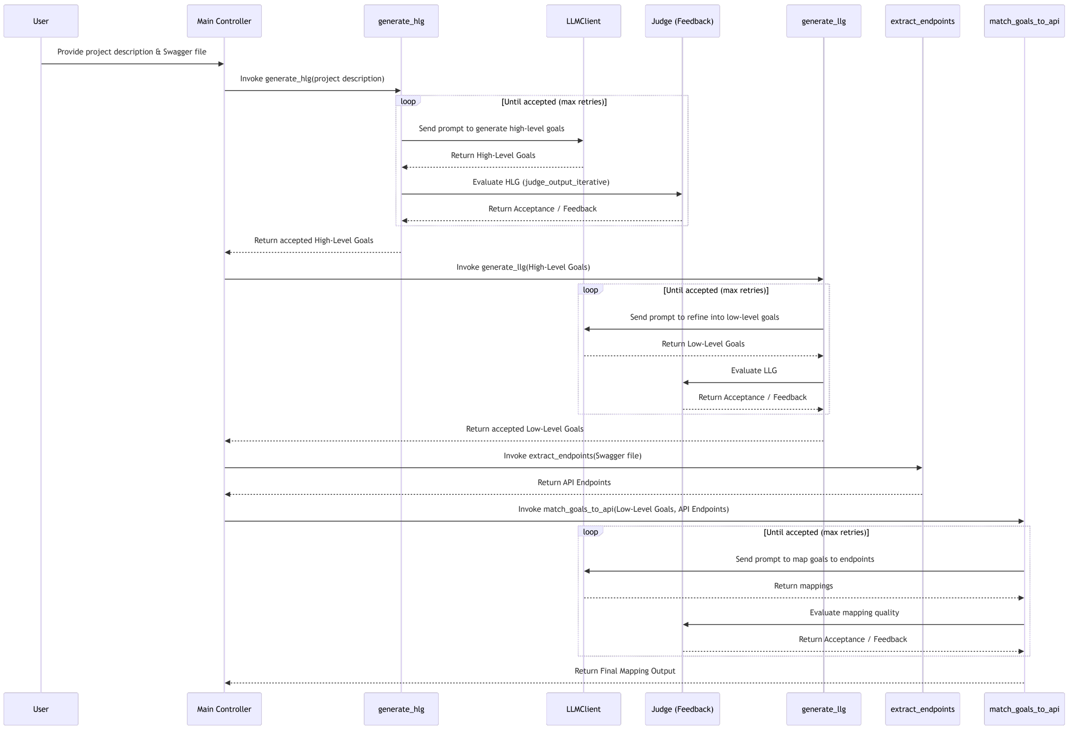


Figure 5.2: Sequence diagram for the Goal-to-API architecture.

is the parameter `response_format`, which is used to ensure the response from the LLM follows a specific structure, when supported by the API.

This feature is known as Structured Outputs, and guides the LLM to return data in a predefined, machine-readable format conforming to a supplied JSON schema, rather than plain natural language text. In AI systems with multi-step workflows, it is often ideal, if not necessary, to extract specific pieces of information from the model's response. Indeed, LLMs can word answers differently each time, which poses a challenge for consistency: if the output is unstructured, each response might require custom parsing or be prone to errors due to variations in phrasing. Requesting a structured output as a JSON with fixed fields makes the response predictable and much easier to handle programmatically, reducing variability and obtaining a stable format that downstream processes can trust.

Pydantic [106] is a Python library that provides facilities for data validation and JSON schema generation using Python type hints. In the context of `call_llm`, Pydantic is used to define the schema of the structured output that is expected from the LLM. A user-defined Pydantic model class inheriting from the `BaseModel` class is created for each kind of structured response needed. Each class lists the fields (and their types) that the LLM's response should contain. Pydantic models act as schemas; they enforce that certain fields are present and of the correct type, can supply default values, and will raise validation errors if the data doesn't match the schema.[107]

After the model responds, Pydantic methods can be used to parse the raw JSON string into a Python object, automatically validating that the model's output conforms to the expected format. If it doesn't (for instance, a field is missing or of the wrong type), Pydantic will throw a validation error, which the code can catch and handle.

In practice, using Pydantic with LLMs typically involves two steps: (1) Including instructions in the prompt, or using provider-specific features and configuration so that the model outputs a JSON object matching the Pydantic schema, and (2) calling `BaseModel.parse_raw(...)` on the model's output to get an instance of the model. If the output is exactly as expected, the parsing succeeds and we get a object of the BaseModel subclass with accessible attributes. If not, it means the model deviated from the format and corrective action can be taken. Pydantic thus serves as a guardrail to enforce consistency and reliability in LLM outputs.

This approach is increasingly common in advanced LLM applications, as it bridges the gap between free-form text generation and the structured data needed by software systems.

OpenAI has recently introduced native support for Structured Outputs [108] [109] parsing. By supplying a Pydantic object to the `response_format` parameter, OpenAI will handle converting the data type to a supported JSON schema and deserializing the JSON response into the typed data structure automatically. Instead of the standard `.create()` call, the `.parse()` method is used, which returns the result already parsed into the Pydantic model instance:

```
from pydantic import BaseModel
from openai import OpenAI

class Step(BaseModel):
    explanation: str
    output: str

class MathResponse(BaseModel):
    steps: list[Step]
    final_answer: str

client = OpenAI()

completion = client.beta.chat.completions.parse(
    model="gpt-4o-2024-08-06",
    messages=[
        {"role": "system", "content": "You are a helpful math tutor."},
        {"role": "user", "content": "solve 8x + 31 = 2"},
    ],
    response_format=MathResponse,
)

message = completion.choices[0].message
if message.parsed:
    print(message.parsed.steps)
    print(message.parsed.final_answer)
else:
    print(message.refusal)
```

This snippet shows the OpenAI client being asked to enforce a structured response that matches the schema defined for a Pydantc model called `OutputModel`. The LLM is essentially instructed behind the scenes to output data conforming to that schema. This also allows for a separation of concerns as we can craft simpler prompts that focus on asking the data that is required, without needing to include specifications on the output schema desired. Then, if successful, the response will have a `.parsed` attribute, to access an already parsed Python object that is an instance of `OutputModel` (via `response.choices[0].message.parsed`). This means the JSON parsing and validation are performed automatically, yielding a Pydantic object (or raising an error if the model's output didn't match the schema).

Similarly, Google allows producing a JSON-formatted output following a specific schema via the `response_schema` configuration parameter, which again can be supplied as a user-defined Pydantic model. If `response_schema` is set, a compatible `responseMimeType`, representing the MIME type of the generated candidate text, must also be set, in particular, `application/json` for JSON response. As for OpenAI, Google Gemini supports native parsing and `.parsed` attribute —again, already structured according to the model defined. [110]

```
from google import genai
from pydantic import BaseModel

class Recipe(BaseModel):
  recipe_name: str
  ingredients: list[str]

client = genai.Client(api_key="GEMINI_API_KEY")
response = client.models.generate_content(
    model='gemini-2.0-flash',
    contents='List a few popular cookie recipes.
    Be sure to include the amounts of ingredients.',
    config={
        'response_mime_type': 'application/json',
        'response_schema': list[Recipe],
    },
)
# Use the response as a JSON string.
print(response.text)

# Use instantiated objects.
my_recipes: list[Recipe] = response.parsed
```

The approach for structured output with Anthropic is different [111], as there appears to be no built-in schema enforcement by the API as the time of writing this. The approach involves including instructions in the prompt for Claude to only output text in JSON format and follow the provided schema, which increases the chance of getting a proper output format. These instructions can be set both via the system message and the user prompt, and, once the response is received as text (`generated_content = response.content[0].text.strip()`), `call_llm` attempts to parse it into the expected response format, an instance of the given Pydantic model, using `response_format.parse_raw(generated_content)`. This method expects that `generated_content` is a valid JSON string conforming to the schema.

If the initial parsing fails (for example, because the response isn't strict JSON), the code implements fallback strategies. The helper function, `extract_json_from_text` is called, which uses a regular expression to look for a JSON-like object in the text. If extraction fails as well, the existing`clean_output()` function is used to "salvage" the raw text. For instance, if JSON parsing fails for a JudgeResult, the output is cleaned and a JudgeResult is built where the cleaned text is assigned to the suggestion field. This way, the system still captures the LLM's feedback even if it wasn't formatted as valid JSON. The same strategy is adopted for the Result response format: the LLM ouput is cleaned and the text is wrapped into the expected format.

The use of Structured Outputs and Pydantic parsing highlights a trend in advanced LLM applications: treating the LLM more like a reliable function or API that returns data, rather than a free-form text generator. By enforcing a schema, the system can validate responses and integrate them into workflows without brittle string processing. For example, in the context of output evaluation, having the model adhere to a schema yields more usable and programmatically analyzable results.

The `call_llm` method also tracks usage statistics including token counts and response time via the `.usage` response field, and logs this data to a `.csv` file, which can be useful for debugging, cost management, and performance optimization purposes.

5.3 Architecture Improvements for Future Work

Although other aspects of the architecture could likely benefit optimization and further enhancement, three main areas for future improvement have been identified: Communication and LLM Interactions, Judge and Feedback, Parsing

5.3.1 Communication and LLM Interactions

The current architecture handles communication among the various phases of the process through a relatively simple mechanism where each action's output is fed into the next stage and feedback updated by concatenating strings.

While this approach is straightforward, the overall flow could benefit from an approach that leverages conversation history across steps. Mainitaining conversation history means preserving the messages (both user prompts and LLM responses) exchanged during each step could be way to of all messages exchanged between steps and feeding them into subsequent interactions in a structured way, as part of a coherent dialogue. When calling the LLM, the `messages` parameter of the request would keep previous interactions along with the current step-specific prompt, rather than sending an isolated prompt and starting a brand-new conversation. As an example, the message history at the low-level goal generation phase would be similar to:

```
    messages = [
    {"role": "system", "content": "You are an expert assistant in
    software requirements analysis."},
    {"role": "user", "content": "Generate high-level goals for
    the following project: ..."},
    {"role": "assistant", "content": "1. Goal A ..."},
    {"role": "user", "content": "Based on
    the high-level goals above, generate low-level goals."},
]
```

By referring back to prior turns in the conversation, this method allows to avoid stuffing every new prompt with redundant context (like repeating all high-level goals), makes prompts more efficient and also helps limiting token usage. Simply passing in high-level goals can make the model miss nuances, whereas if the model "sees" that those goals were derived from a specific project description, more relevant low-level goals can be generated. This approach ensures the LLM has full context and produces more consistent outputs by keeping the whole chain of thoughts and actions. Moreover, the judge-feedback loop would also benefit from the conversation, as each suggestion or correction can be grounded in prior messages and the LLM can "see" what it previously said and why it's now being asked to improve.

5.3.2 Judge and Feedback

The logic for achieving an improved output of the domain-specific actions can be enhanced in several ways. As an example, in this implementation, the feedback-retry mechanism iteratively calls the domain-specific action appending the suggestion for improvement to the prompt. Across iterations, new suggestions are concatenated with suggestions from previous iterations. This can lead to issues related to growth of prompt length, complexity and token usage, even if careful setting of the number of retries limits the problem. It could also lead to repetitive sentences in the prompt, as suggestions across different iterations could be similar and somewhat overlapping. An optimization of the update strategy might therefore be necessary; for example, instead of relying on raw string concatenation, a more structured feedback format could be leveraged (e.g., JSON), which could also allow for a more nuanced approach, e.g. prioritizing recent over older feedback, more severe over softer comments. Additionally, in this setting where the domain-specific action is iteratively called, the refined version of the output is generated based on the original function prompt, simply enriched with the generated improvement suggestion, while it could be useful to provide previous output as well to serve as a baseline upon which to improve. This would help avoid generating a refined output which might actually be similar to the original output. As previously suggested, one way to optimize the flow would be storing conversation history. On the other hand, the current implementation of the judge mechanism heavily relies on the chosen approach with score assignment, motivation and improvement suggestion, plus the evaluation criteria and thresholds are hard-coded within the prompts. This reduces flexibility if different evaluation methods or standards are preferred, so future work could explore ways to optimize the judge phase at both levels.

5.3.3 LLM System Instruction and Response Format

All providers allow specifying a system instruction in the model request, which serves to prime the assistant with certain behavior or context. This instruction can be further customized to tailor the model to better meet specific guidelines and desired behaviour.

Regarding the Structured Outputs part, the user-defined Pydantic models can be further optimized. As an example, the current implementation for the domain-specific functions uses a response format based on a single `str` field, which doesn't take full advantage of the output schema enforcement possibilities.

```
class Result(BaseModel):
 output: str = Field(description = "The result of the action")
```

For functions like `generate_hlg`, `generate_llg`, which essentially require the generation of a list of goals, something as below could be more appropriate:

```
class Goals(BaseModel):
 output: List[str] = Field(descrpition = "List of goals representing functionalities
 expected by the end-user using the application")
```

Again, being more precise in the response format definition enhances consistency in the outputs across different functions and runs. It is also worth mentioning that the Instructor Python library could offer a way to leverage Structured Outputs with Anthropic as well [112], while also being applicable to other models including OpenAI. The application of Pydantic + Instructor libraries combined is therefore worth further investigation.[113] [114]

5.3.4 Parsing and Error Handling

While the current implementation for parsing and validation handles various cases, further interventions could enhance robustness, clarity, and maintainability. For instance, isolating functionalities —such as CSV logging, usage information extraction, and fallback parsing strategies— into dedicated helper functions would reduce duplication of code snippets and improve readability and keep concerns separated from the LLM client logic.

Additionally, the error handling and logging mechanisms could be improved by integrating a proper logging framework rather than relying on print statements. This would allow for different log levels and enhanced control, making debugging easier when issues arise. It would also be worthwhile to consider adding a retry mechanism for transient errors due to temporary API failures.

The regex used for extracting JSON might also be refined, for example to work better when the output contains additional text beyond the JSON object. The fallback strategies could be further optimized as well to ensure more robust and reliable code. This layered approach would help in retaining as much useful information from the LLM output as possible, even when it deviates from the expected format. Careful LLM mocking to trigger and test unwanted scenarios would further ensure that the parsing logic remains strong under diverse conditions. Regarding usage tracking logic, the code currently generally assumes that usage metadata is always available, therefore, introducing checks and assertions to verify that it has been correctly assigned, or default values for cases where it might be missing, would help make the code more robust.

Chapter 6

Conclusions

The thesis brings the following contributions: The exploratory review of state-of-the-art approaches in the field provides an extensive overview on the practical challenges and benefits of applying LLMs to automate the core tasks of requirements engineering. The experiment on Goal-to-API mapping using ChatGPT offers practical insights into a simple application of the GPT model to the tasks of eliciting and classifying software requirements and mapping them to API endpoints via prompts and conversational interactions.

The introduction of an LLM-based architecture for Goal-to-API mapping offers a structured, modular solution that integrates multiple LLM providers, implements an iterative refinement mechanism to enhance quality of the generated results and leverages structured outputs to ensure consistency. This architectural approach not only contributes to streamline the mapping process but also highlights avenues for further research.

Overall, this research advances understanding of how LLMs can address longstanding challenges in Requirements Engineering and explores the promise and current limitations of LLM-based approaches.

The combined findings advocate for further exploration into LLM-based solutions, especially the exploration of agent and multi-agent-based systems and more refined LLM interactions, aiming to achieve higher consistency, accuracy, and scalability in automating requirements engineering processes.

Collectively, these contributions underline the promise of integrating Large Language Models into Requirements Engineering tasks —from elicitation and classification to traceability and API recommendation—, marking a significant step forward in automating and enhancing requirements management.

Bibliography

[1] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. «Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges». In: *ACM Comput. Surv.* 53.3 (June 2020). ISSN: 0360-0300. DOI: 10.1145/3383458. URL: https://doi.org/10.1145/3383458 (cit. on p. 3).

[2] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.* 2014. arXiv: 1406.1078 [cs.CL] (cit. on p. 4).

[3] url: https://ai.plainenglish.io/vanishing-gradient-problem-in-rnns-d362235005c (cit. on p. 4).

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate.* 2016. arXiv: 1409.0473 [cs.CL] (cit. on pp. 5, 8).

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. «Attention is all you need». In: *Advances in neural information processing systems* 30 (2017) (cit. on p. 6).

[6] Shashank Sonkar and Richard G. Baraniuk. *Investigating the Role of Feed-Forward Networks in Transformers Using Parallel Attention and Feed-Forward Net Design.* 2023. arXiv: 2305.13297 [cs.CL] (cit. on p. 7).

[7] Wayne Xin Zhao et al. *A Survey of Large Language Models.* 2023. arXiv: 2303.18223 [cs.CL] (cit. on pp. 8–10).

[8] Xinyi Hou et al. *Large Language Models for Software Engineering: A Systematic Literature Review.* 2024. arXiv: 2308.10620 [cs.SE] (cit. on pp. 9, 10).

[9] Zibin Zheng, Kaiwen Ning, Jiachi Chen, Yanlin Wang, Wenqing Chen, Lianghong Guo, and Weicheng Wang. *Towards an Understanding of Large Language Models in Software Engineering Tasks.* 2023. arXiv: 2308.11396 [cs.SE] (cit. on p. 9).

[10] Krishna Ronanki, Christian Berger, and Jennifer Horkoff. «Investigating ChatGPT's Potential to Assist in Requirements Elicitation Processes». In: *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).* 2023, pp. 354–361. DOI: 10.1109/SEAA60479.2023.00061 (cit. on pp. 14, 24).

[11] Binnur Görer and Fatma Başak Aydemir. «Generating Requirements Elicitation Interview Scripts with Large Language Models». In: *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW).* 2023, pp. 44–51. DOI: 10.1109/REW57809.2023.00015 (cit. on pp. 14, 15, 24).

[12] Yi Yang, Qingwen Zhang, Ci Li, Daniel Simões Marta, Nazre Batool, and John Folkesson. «Human-centric autonomous systems with llms for user command reasoning». In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2024, pp. 988–994 (cit. on pp. 14, 17, 24).

[13] Janosch Luttmer, Vitalijs Prihodko, Dominik Ehring, and Arun Nagarajah. «Requirements extraction from engineering standards–systematic evaluation of extraction techniques». In: *Procedia CIRP* 119 (2023), pp. 794–799 (cit. on pp. 14, 19, 24).

[14] Adailton Ferreira de Araújo and Ricardo Marcondes Marcacini. «Re-bert: automatic extraction of software requirements from app reviews using bert language model». In: *Proceedings of the 36th annual ACM symposium on applied computing*. 2021, pp. 1321–1327 (cit. on pp. 14, 20, 25).

[15] Yawen Wang, Junjie Wang, Hongyu Zhang, Kairui Wang, and Qing Wang. «What are Pros and Cons? Stance Detection and Summarization on Feature Request». In: *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE. 2023, pp. 1–12 (cit. on pp. 14, 22, 25).

[16] Tobias Hey, Jan Keim, Anne Koziolek, and Walter F Tichy. «Norbert: Transfer learning for requirements classification». In: *2020 IEEE 28th international requirements engineering conference (RE)*. IEEE. 2020, pp. 169–179 (cit. on pp. 14, 26, 35, 39).

[17] Gang Li, Chengpeng Zheng, Min Li, and Haosen Wang. «Automatic requirements classification based on graph attention network». In: *IEEE Access* 10 (2022), pp. 30080–30090 (cit. on pp. 14, 27, 39).

[18] Bing Li and Xiuwen Nong. «Automatically classifying non-functional requirements using deep neural network». In: *Pattern Recognition* 132 (2022), p. 108948 (cit. on pp. 14, 28, 39).

[19] Archana Tikayat Ray, Bjorn F Cole, Olivia J Pinon Fischer, Ryan T White, and Dimitri N Mavris. «aerobert-classifier: Classification of aerospace requirements using bert». In: *Aerospace* 10.3 (2023), p. 279 (cit. on pp. 14, 29, 32, 40).

[20] Archana Tikayat Ray, Bjorn F. Cole, Olivia J. Pinon Fischer, Anirudh Prabhakara Bhat, Ryan T. White, and Dimitri N. Mavris. «Agile Methodology for the Standardization of Engineering Requirements Using Large Language Models». In: *Systems* 11.7 (2023). ISSN: 2079-8954. DOI: `10.3390/systems11070352`. URL: `https://www.mdpi.com/2079-8954/11/7/352` (cit. on pp. 14, 31, 40).

[21] Waad Alhoshan, Alessio Ferrari, and Liping Zhao. «Zero-shot learning for requirements classification: An exploratory study». In: *Information and Software Technology* 159 (2023), p. 107202 (cit. on pp. 14, 33, 40).

[22] Xianchang Luo, Yinxing Xue, Zhenchang Xing, and Jiamou Sun. «PRCBERT: Prompt Learning for Requirement Classification using BERT-based Pretrained Language Models». In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. Rochester, MI, USA: Association for Computing Machinery, 2023. ISBN: 9781450394758. DOI: `10.1145/3551349.3560417`. URL: `https://doi.org/10.1145/3551349.3560417` (cit. on pp. 14, 35, 40).

[23] Abdelkarim El-Hajjami, Nicolas Fafin, and Camille Salinesi. «Which AI Technique Is Better to Classify Requirements? An Experiment with SVM, LSTM, and ChatGPT». In: *arXiv preprint arXiv:2311.11547* (2023) (cit. on pp. 14, 36, 41).

[24] Georgia Xanthopoulou, Miltiadis Siavvas, Ilias Kalouptsoglou, Dionysios Kehagias, and Dimitrios Tzovaras. *Software Requirements Classification: From Bag-of-Words to Transformer*. June 2024. DOI: `10.13140/RG.2.2.35146.58563` (cit. on pp. 14, 38, 41).

[25] Saad Ezzini, Sallam Abualhaija, Chetan Arora, and Mehrdad Sabetzadeh. «Automated handling of anaphoric ambiguity in requirements: a multi-solution study». In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 187–199 (cit. on pp. 14, 42, 44, 51).

[26] Sanaz Mohammad Jafari, Savas Yildirim, Mucahit Cevik, and Ayse Basar. «Anaphoric Ambiguity Resolution in Software Requirement Texts». In: *2023 IEEE International Conference on Big Data (BigData)*. IEEE. 2023, pp. 4722–4730 (cit. on pp. 14, 43, 51).

[27] Robert K Helmeczi, Mucahit Cevik, and Savas Yıldırım. «A Prompt-based Few-shot Learning Approach to Software Conflict Detection». In: *arXiv preprint arXiv:2211.02709* (2022) (cit. on pp. 14, 44, 52).

[28] Ambarish Moharil and Arpit Sharma. «Tabasco: A transformer based contextualization toolkit». In: *Science of Computer Programming* 230 (2023), p. 102994 (cit. on pp. 14, 45, 52).

[29] Reena S Satpute and Avinash Agrawal. «Machine Learning Approach for Ambiguity Detection in Social Media Context». In: *2023 International Conference on Communication, Security and Artificial Intelligence (ICCSAI)*. IEEE. 2023, pp. 516–522 (cit. on pp. 14, 46, 52).

[30] Alessandro Fantechi, Stefania Gnesi, Laura Semini, et al. «Rule-based NLP vs ChatGPT in ambiguity detection, a preliminary study.» In: *REFSQ Workshops*. 2023 (cit. on pp. 14, 47, 52).

[31] Dipeeka Luitel, Shabnam Hassani, and Mehrdad Sabetzadeh. «Improving requirements completeness: automated assistance through large language models». In: *Requirements Engineering* 29.1 (2024), pp. 73–95. ISSN: 1432-010X. DOI: `10.1007/s00766-024-00416-3`. URL: `https://doi.org/10.1007/s00766-024-00416-3` (cit. on pp. 14, 48, 49, 52).

[32] Jinfeng Lin, Yalin Liu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang. «Traceability Transformed: Generating More Accurate Links with Pre-Trained BERT Models». In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021, pp. 324–335. DOI: `10.1109/ICSE43902.2021.00040` (cit. on pp. 14, 54, 69, 73).

[33] Jinfeng Lin, Yalin Liu, and Jane Cleland-Huang. «Information retrieval versus deep learning approaches for generating traceability links in bilingual projects». In: *Empirical Software Engineering* 27.1 (2022), p. 5 (cit. on pp. 14, 56, 73).

[34] Kai Xia and JinHua Wang. «Entry traceability technology based on requirement knowledge graph». In: *International Conference on Cloud Computing, Internet of Things, and Computer Applications (CICA 2022)*. Vol. 12303. SPIE. 2022, pp. 338–345 (cit. on pp. 14, 57, 73).

[35] Jiahao Tian, Li Zhang, and Xiaoli Lian. «A cross-level requirement trace link update model based on bidirectional encoder representations from transformers». In: *Mathematics* 11.3 (2023), p. 623 (cit. on pp. 14, 54, 58, 73).

[36] Ali Majidzadeh, Mehrdad Ashtiani, and Morteza Zakeri-Nasrabadi. «Multi-type requirements traceability prediction by code data augmentation and fine-tuning MS-CodeBERT». In: *Computer Standards & Interfaces* 90 (2024), p. 103850. ISSN: 0920-5489. DOI: `https://doi.org/10.1016/j.csi.2024.103850`. URL: `https://www.sciencedirect.com/science/article/pii/S0920548924000199` (cit. on pp. 14, 60, 73).

[37] Jinpeng Lan, Lina Gong, Jingxuan Zhang, and Haoxiang Zhang. «BTLink: automatic link recovery between issues and commits based on pre-trained BERT model». In: *Empirical Software Engineering* 28.4 (2023), p. 103 (cit. on pp. 14, 53, 62, 74).

[38] Yang Deng, Bangchao Wang, Qiang Zhu, Junping Liu, Jiewen Kuang, and Xingfu Li. «MTLink: Adaptive multi-task learning based pre-trained language model for traceability link recovery between issues and commits». In: *Journal of King Saud University-Computer and Information Sciences* 36.2 (2024), p. 101958 (cit. on pp. 14, 66, 74).

[39] Jianfei Zhu, Guanping Xiao, Zheng Zheng, and Yulei Sui. «Deep semi-supervised learning for recovering traceability links between issues and commits». In: *Journal of Systems and Software* (2024), p. 112109 (cit. on pp. 14, 68, 74).

[40] Alberto D. Rodriguez, Katherine R. Dearstyne, and Jane Cleland-Huang. «Prompts Matter: Insights and Strategies for Prompt Engineering in Automated Software Traceability». In: *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*. 2023, pp. 455–464. DOI: `10.1109/REW57809.2023.00087` (cit. on pp. 14, 71, 74).

[41] Jameleddine Hassine. «An LLM-based Approach to Recover Traceability Links between Security Requirements and Goal Models». In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. EASE '24. Salerno, Italy: Association for Computing Machinery, 2024, pp. 643–651. ISBN: 9798400717017. DOI: `10.1145/3661167.3661261`. URL: `https://doi.org/10.1145/3661167.3661261` (cit. on pp. 14, 53, 72, 74).

[42] Moshi Wei, Nima Shiri Harzevili, Yuchao Huang, Junjie Wang, and Song Wang. «CLEAR: <u>c</u>ontrastive <u>le</u>arning for <u>A</u>PI <u>r</u>ecommendation». In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 376–387. ISBN: 9781450392211. DOI: `10.1145/3510003.3510159`. URL: `https://doi.org/10.1145/3510003.3510159` (cit. on pp. 14, 75, 91).

[43] Ivana Clairine Irsan, Ting Zhang, Ferdian Thung, Kisub Kim, and David Lo. «Picaso: enhancing api recommendations with relevant stack overflow posts». In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE. 2023, pp. 92–37 (cit. on pp. 14, 76, 92).

[44] Qing Huang, Zhenyu Wan, Zhenchang Xing, Changjing Wang, Jieshan Chen, Xiwei Xu, and Qinghua Lu. «Let's Chat to Find the APIs: Connecting Human, LLM and Knowledge Graph through AI Chain». In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 471–483 (cit. on pp. 78, 92).

[45] Yafeng Gu, Yiheng Shen, Xiang Chen, Shaoyu Yang, Yiling Huang, and Zhixiang Cao. «APICom: Automatic API Completion via Prompt Learning and Adversarial Training-based Data Augmentation». In: *Proceedings of the 14th Asia-Pacific Symposium on Internetware*. 2023, pp. 259–269 (cit. on pp. 14, 78, 92).

[46] Ye Wang, Junwu Chen, Qiao Huang, Xin Xia, and Bo Jiang. «Deep learning-based open api recommendation for mashup development». In: *Science China Information Sciences* 66.7 (2023), p. 172102 (cit. on pp. 14, 80, 92).

[47] Chunxiang Zhang, Shaowei Qin, Hao Wu, and Lei Zhang. «Cooperative Mashup Embedding Leveraging Knowledge Graph for Web API Recommendation». In: *IEEE Access* (2024) (cit. on pp. 14, 81, 93).

[48] Keqi Li, Xingli Tang, Fenghang Li, Hui Zhou, Chunyang Ye, and Wenyu Zhang. «PyBartRec: Python API Recommendation with Semantic Information». In: *Proceedings of the 14th Asia-Pacific Symposium on Internetware*. 2023, pp. 33–43 (cit. on pp. 14, 83, 93).

[49] Zhihao Li et al. «PTM-APIRec: Leveraging Pre-trained Models of Source Code in API Recommendation». In: *ACM Trans. Softw. Eng. Methodol.* 33.3 (Mar. 2024). ISSN: 1049-331X. DOI: `10.1145/3632745`. URL: `https://doi.org/10.1145/3632745` (cit. on pp. 14, 84, 93).

[50] Zexiong Ma, Shengnan An, Bing Xie, and Zeqi Lin. «Compositional API Recommendation for Library-Oriented Code Generation». In: *2024 IEEE/ACM 32nd International Conference on Program Comprehension (ICPC)*. 2024, pp. 87–98 (cit. on pp. 14, 88, 94).

[51] Jun Chen, Lei Xiao, and Yuxiang Shen. «API Completion Recommendation Algorithm Based on Programming Site Context». In: *2024 10th International Symposium on System Security, Safety, and Reliability (ISSSR)*. IEEE. 2024, pp. 251–257 (cit. on pp. 14, 89, 94).

[52] Fabiano Dalpiaz, Davide Dell'Anna, Fatma Basak Aydemir, and Sercan Çevikol. «Requirements classification with interpretable machine learning and dependency parsing». In: *2019 IEEE 27th International Requirements Engineering Conference (RE)*. IEEE. 2019, pp. 142–152 (cit. on pp. 26, 39).

[53] Zijad Kurtanović and Walid Maalej. «Automatically classifying functional and non-functional requirements using supervised machine learning». In: *2017 IEEE 25th International Requirements Engineering Conference (RE)*. Ieee. 2017, pp. 490–495 (cit. on pp. 26, 39).

[54] Zahra Shakeri Hossein Abad, Oliver Karras, Parisa Ghazi, Martin Glinz, Guenther Ruhe, and Kurt Schneider. «What works better? a study of classifying requirements». In: *2017 IEEE 25th International Requirements Engineering Conference (RE)*. IEEE. 2017, pp. 496–501 (cit. on pp. 26, 39).

[55] Alex Dekhtyar and Vivian Fong. «Re data challenge: Requirements identification with word2vec and tensorflow». In: *2017 IEEE 25th International Requirements Engineering Conference (RE)*. IEEE. 2017, pp. 484–489 (cit. on pp. 27, 39).

[56] Wenpeng Yin, Jamaal Hay, and Dan Roth. «Benchmarking zero-shot text classification: Datasets, evaluation and entailment approach». In: *arXiv preprint arXiv:1909.00161* (2019) (cit. on p. 30).

[57] Alan Akbik, Duncan Blythe, and Roland Vollgraf. «Contextual string embeddings for sequence labeling». In: *Proceedings of the 27th international conference on computational linguistics*. 2018, pp. 1638–1649 (cit. on p. 32).

[58] Michael Riesener, Christian Dölle, Annika Becker, Sofia Gorbatcheva, Eric Rebentisch, and Günther Schuh. «Application of natural language processing for systematic requirement management in model-based systems engineering». In: *INCOSE International Symposium*. Vol. 31. 1. Wiley Online Library. 2021, pp. 806–815 (cit. on p. 32).

[59] Muideen Ajagbe and Liping Zhao. «Retraining a bert model for transfer learning in requirements engineering: A preliminary study». In: *2022 IEEE 30th International Requirements Engineering Conference (RE)*. IEEE. 2022, pp. 309–315 (cit. on p. 33).

[60] Jeniya Tabassum, Mounica Maddela, Wei Xu, and Alan Ritter. «Code and named entity recognition in stackoverflow». In: *arXiv preprint arXiv:2005.01634* (2020) (cit. on p. 33).

[61] Garima Malik, Mucahit Cevik, Devang Parikh, and Ayse Basar. «Identifying the requirement conflicts in SRS documents using transformer-based sentence embeddings». In: *arXiv preprint arXiv:2206.13690* (2022) (cit. on p. 44).

[62] Timo Schick and Hinrich Schütze. «Exploiting cloze questions for few shot text classification and natural language inference». In: *arXiv preprint arXiv:2001.07676* (2020) (cit. on p. 44).

[63] Giuseppe Lami. *QuARS: A tool for analyzing requirements.* Carnegie Mellon University, Software Engineering Institute, 2005 (cit. on pp. 47, 52).

[64] Alberto Fernández et al. «Cost-sensitive learning». In: *Learning from imbalanced data sets* (2018), pp. 63–78 (cit. on pp. 50, 53).

[65] Gerard Salton, Anita Wong, and Chung-Shu Yang. «A vector space model for automatic indexing». In: *Communications of the ACM* 18.11 (1975), pp. 613–620 (cit. on p. 53).

[66] Andrea De Lucia, Rocco Oliveto, and Genoveffa Tortora. «Assessing IR-based traceability recovery tools through controlled experiments». In: *Empirical Software Engineering* 14 (2009), pp. 57–92 (cit. on p. 53).

[67] Alex Dekhtyar, Jane Huffman Hayes, Senthil Sundaram, Ashlee Holbrook, and Olga Dekhtyar. «Technique integration for requirements assessment». In: *15th IEEE International Requirements Engineering Conference (RE 2007).* IEEE. 2007, pp. 141–150 (cit. on p. 53).

[68] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. «Semantically enhanced software traceability using deep learning techniques». In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* IEEE. 2017, pp. 3–14 (cit. on p. 53).

[69] Zhiheng Huang, Wei Xu, and Kai Yu. «Bidirectional LSTM-CRF models for sequence tagging». In: *arXiv preprint arXiv:1508.01991* (2015) (cit. on p. 58).

[70] Daniel Blasco, Carlos Cetina, and Óscar Pastor. «A fine-grained requirement traceability evolutionary algorithm: Kromaia, a commercial video game case study». In: *Information and Software Technology* 119 (2020), p. 106235 (cit. on p. 61).

[71] Yan Sun, Qing Wang, and Ye Yang. «Frlink: Improving the recovery of missing issue-commit links by revisiting file relevance». In: *Information and Software Technology* 84 (2017), pp. 33–47 (cit. on p. 64).

[72] Hang Ruan, Bihuan Chen, Xin Peng, and Wenyun Zhao. «DeepLink: Recovering issue-commit links based on deep learning». In: *Journal of Systems and Software* 158 (2019), p. 110406 (cit. on p. 64).

[73] Pooya Rostami Mazrae, Maliheh Izadi, and Abbas Heydarnoori. «Automated recovery of issue-commit links leveraging both textual and non-textual data». In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE. 2021, pp. 263–273 (cit. on p. 64).

[74] Chenyuan Zhang, Yanlin Wang, Zhao Wei, Yong Xu, Juhong Wang, Hui Li, and Rongrong Ji. «EALink: An efficient and accurate pre-trained framework for issue-commit link recovery». In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE. 2023, pp. 217–229 (cit. on pp. 64, 74).

[75] Xiao Liu, Fanjin Zhang, Zhenyu Hou, Li Mian, Zhaoyu Wang, Jing Zhang, and Jie Tang. «Self-supervised learning: Generative or contrastive». In: *IEEE transactions on knowledge and data engineering* 35.1 (2021), pp. 857–876 (cit. on p. 66).

[76] Shikun Liu, Edward Johns, and Andrew J Davison. «End-to-end multi-task learning with attention». In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition.* 2019, pp. 1871–1880 (cit. on p. 68).

[77] Dong-Hyun Lee et al. «Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks». In: *Workshop on challenges in representation learning, ICML.* Vol. 3. 2. Atlanta. 2013, p. 896 (cit. on p. 69).

[78] Ahmet Iscen, Giorgos Tolias, Yannis Avrithis, and Ondrej Chum. «Label propagation for deep semi-supervised learning». In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 5070–5079 (cit. on p. 69).

[79] Jianfei Zhu, Guanping Xiao, Zheng Zheng, and Yulei Sui. «Enhancing traceability link recovery with unlabeled data». In: *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2022, pp. 446–457 (cit. on p. 70).

[80] Vahdat Abdelzad, Daniel Amyot, and Timothy C Lethbridge. «Adding a textual syntax to an existing graphical modeling language: Experience report with GRL». In: *SDL 2015: Model-Driven Engineering for Smart Cities: 17th International SDL Forum, Berlin, Germany, October 12-14, 2015, Proceedings 17*. Springer. 2015, pp. 159–174 (cit. on p. 72).

[81] Yun Peng, Shuqing Li, Wenwei Gu, Yichen Li, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. «Revisiting, Benchmarking and Exploring API Recommendation: How Far Are We?» In: *IEEE Transactions on Software Engineering* 49.4 (2023), pp. 1876–1897. DOI: 10.1109/TSE.2022.3197063 (cit. on p. 75).

[82] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. «API method recommendation without worrying about the task-API knowledge gap». In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 293–304 (cit. on p. 75).

[83] Mohammad Masudur Rahman, Chanchal K Roy, and David Lo. «Rack: Automatic api recommendation using crowdsourced knowledge». In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 349–359 (cit. on p. 75).

[84] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. «Deep API learning». In: *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 2016, pp. 631–642 (cit. on p. 75).

[85] Jinpei Yan, Yong Qi, Qifan Rao, Hui He, et al. «Learning API Suggestion via Single LSTM Network with Deterministic Negative Sampling.» In: *SEKE*. 2018, pp. 137–136 (cit. on p. 75).

[86] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. «Representation learning with contrastive predictive coding». In: *arXiv preprint arXiv:1807.03748* (2018) (cit. on p. 75).

[87] Qing Huang, Zishuai Li, Zhenchang Xing, Zhengkang Zuo, Xin Peng, Xiwei Xu, and Qinghua Lu. «Answering uncertain, under-specified api queries assisted by knowledge-aware human-ai dialogue». In: *IEEE Transactions on Software Engineering* (2023) (cit. on p. 78).

[88] Xincheng He, Lei Xu, Xiangyu Zhang, Rui Hao, Yang Feng, and Baowen Xu. «Pyart: Python api recommendation in real-time». In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 1634–1645 (cit. on p. 83).

[89] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. «Spt-code: Sequence-to-sequence pre-training for learning source code representations». In: *Proceedings of the 44th international conference on software engineering*. 2022, pp. 2006–2018 (cit. on p. 83).

[90] Yuning Kang, Zan Wang, Hongyu Zhang, Junjie Chen, and Hanmo You. «Apirecx: Cross-library api recommendation via pre-trained language model». In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 2021, pp. 3425–3436 (cit. on p. 85).

[91] Ryan Greene, Ted Sanders, Lilian Weng, and Arvind Neelakantan. «New and improved embedding model». In: *OpenAI Blog. Available online: https://openai. com/blog/new-and-improved-embedding-model (accessed on 28 November 2023)* (2022) (cit. on p. 89).

[92] Robert Feldt and Riccardo Coppola. *Semantic API Alignment: Linking High-level User Goals to APIs*. 2024. arXiv: 2405.04236 [cs.SE]. URL: https://arxiv.org/abs/2405.04236 (cit. on pp. 95, 96).

[93] *APIExperimentSheet*. URL: https://docs.google.com/spreadsheets/d/1-SwUV0vSmdQ3p0okm6JKmKdHbe1rLvJE/edit?usp=sharing&ouid=110395767446024448698&rtpof=true&sd=true (cit. on p. 95).

[94] *Arcuri EMB*. URL: https://github.com/WebFuzzing/EMB/tree/master/openapi-swagger (cit. on p. 95).

[95] *Genome Nexus*. URL: https://github.com/genome-nexus/genome-nexus (cit. on p. 105).

[96] *Features-Service*. URL: https://github.com/JavierMF/features-service (cit. on p. 105).

[97] *Market*. URL: https://github.com/aleksey-lukyanets/market (cit. on p. 105).

[98] *Reservations-API*. URL: https://github.com/cyrilgavala/reservations-api (cit. on p. 105).

[99] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. *ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design*. 2023. arXiv: 2303.07839 [cs.SE]. URL: https://arxiv.org/abs/2303.07839 (cit. on p. 110).

[100] *Goal-to-API-Mapping Architecture*. URL: https://github.com/vittoriaocl/Goal-to-API-Mapping.git (cit. on p. 111).

[101] *OpenAI API*. URL: https://platform.openai.com/docs/api-reference/introduction (cit. on p. 111).

[102] *Geimini API*. URL: https://ai.google.dev/gemini-api/docs/get-started/tutorial?hl=it&lang=python (cit. on p. 111).

[103] *Anthropic API*. URL: https://docs.anthropic.com/it/api/client-sdks (cit. on p. 111).

[104] *Understanding LLM-as-a-Judge: The Future of Automated Evaluation*. URL: https://shashankguda.medium.com/understanding-llm-as-a-judge-the-future-of-automated-evaluation-b124bd178fb1 (cit. on p. 113).

[105] *Using LLM-as-a-judge*. URL: https://huggingface.co/learn/cookbook/llm_judge (cit. on p. 113).

[106] *Pydantic*. URL: https://docs.pydantic.dev/latest/ (cit. on p. 115).

[107] *Controlling Large Language Model Output with Pydantic*. URL: https://medium.com/@mattchinnock/controlling-large-language-model-output-with-pydantic-74b2af5e79d1 (cit. on p. 115).

[108] *Introducing Structured Outputs in the API*. URL: https://openai.com/index/introducing-structured-outputs-in-the-api/ (cit. on p. 116).

[109] *Introduction to Structured Outputs*. URL: https://cookbook.openai.com/examples/structured_outputs_intro (cit. on p. 116).

[110] *Generate structured output with the Gemini API.* URL: https://ai.google.dev/gemini-api/docs/structured-output?lang=python (cit. on p. 117).

[111] *Increase output consistency (JSON mode).* URL: https://docs.anthropic.com/en/docs/test-and-evaluate/strengthen-guardrails/increase-consistency (cit. on p. 117).

[112] *Structured Outputs and Prompt Caching with Anthropic.* URL: https://python.useinstructor.com/blog/2024/10/23/structured-outputs-and-prompt-caching-with-anthropic/ (cit. on p. 119).

[113] *How to Get Structured Output from LLMs Applications Using Pydantic and Instrutor.* URL: https://medium.com/@pedro.aquino.se/how-to-get-structured-output-from-llms-applications-using-pydantic-and-instrutor-87d237c03073 (cit. on p. 119).

[114] *Instructor, The Most Popular Library for Simple Structured Outputs.* URL: https://python.useinstructor.com/ (cit. on p. 119).