Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

MASTERTHESIS

Improving Large Language Models in Repository Level Programming through Self-Alignment and Retrieval-Augmented Generation

submitted by

Jan Strich

Faculty of Mathematics, Computer Science and Natural Sciences

Department of Computer Science

Language Technology (LT) group

Degree: M. Sc. Computer Science

Matrikelnummer: 6749619

Submission date: 02.06.2024

First Examiner: Prof. Dr. Chris Biemann

Second Examiner: Dr. Irina Nikishina

Supervisor: Florian Schneider

Abstract

Repository-level programming involves writing code specific to a particular domain or project. Large language models (LLMs) such as ChatGPT, GitHub Copilot, Llama, or Mistral can assist programmers as coding assistants and knowledge sources to make the coding process faster and more efficient. This thesis aims to improve coding assistants performance by implementing a Self-Alignment process and a retrieval-augmented generation (RAG) pipeline for a specific repository. Self-Alignment is the process of creating a training dataset by an LLM, curating the samples to improve the dataset quality and supervised fine-tuning with the curated dataset. In comparison, RAG pipelines use a vector database to fetch relevant documents from the repository using similarity search and provide them as context into the model.

This thesis introduces SpyderCodeQA, a dataset that tests the ability of models to understand the source code, the dependencies between files, and the overall meta-information about the repository. To evaluate the fine-tuned LLM and RAG pipeline on SpyderCodeQA, the LLM-as-a-Judge evaluation is used, which compares the models pairwise with GPT-3.5 as judge. The results show that models that the fine-tuned LLM and RAG pipelines outperform the LLM without adjustment on the SpyderCodeQA. In addition, the results show that combining both approaches leads to an interaction effect that further improves SpyderCodeQA's performance. Further ablation studies are conducted investigating hyperparameters such as Top-P, Temperature and the choice of judge. A qualitative analysis of the evaluation results is carried out in order to better understand the effects. The source code and the dataset can be found on here[1].

[1] https://anonymous.4open.science/r/ma_llm-382D

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

Large Language Models (LLMs) have transformed the programming world with their exceptional ability to produce code that adheres to human directives and assists in the process. The emergence of these models has led to a new era of artificial intelligence (AI), revolutionizing how we work and impacting our daily routines. General-purpose LLMs such as ChatGPT[1], Gemini[2], and Claude 3[3], along with open-source models like Llama 2 [1], have gained widespread recognition. Additionally, task-specific LLMs have been developed for summarizing text, generating questions [2], and even performing coding-specific tasks such as coding summarization [3], coding generation [4, 5], answering questions regarding source code [6] or generating code documentation on repository-level [7]. In this thesis, these models, defined as coding assistants, are invaluable to any programming team. Developing software applications, games, or machine learning models involves writing code using programming languages. However, coding assistants can help better understand how the code works, generate code and fix errors faster. With 92% of developers in the US [4] currently utilizing AI coding assistants, it's clear that it is relevant to improve them further.

Code suggestions in the IDE are effectively provided by the commercial AI-assisted coding Chatbot GitHub Copilot[5], which is used for code completion and as a coding assistant. Other alternatives like Codeium[6], Cursor[7] or Starcoder [8] provided similar features. Developers benefit from these assistants by experiencing increased productivity, reduced debugging time, and improved code quality. Although coding assistants have undeniable advantages, their usage raises specific concerns and challenges. Coding assistants generally assist with an existing codebase, generate code, or answer questions.

In this thesis, the task of assisting with a specific project organized in a Git repository is defined as repository-level programming. Developers benefit from coding assistants at this level, as becoming familiar with a new project or repository can be time-consuming and challenging. However, it is important to note that coding assistants may generate incorrect information, also known as hallucinations, par-

[1] https://chat.openai.com
[2] https://gemini.google.com/app
[3] https://www.anthropic.com/claude
[4] https://is.gd/KQcT5L
[5] https://github.com/features/copilot/
[6] https://www.codium.ai
[7] https://cursor.sh

ticularly when requests go beyond the model's training data or require additional knowledge [9]. Another issue associated with coding assistants is data protection. Providing coding suggestions requires access to a significant amount of code and data, which users may need to share with the commercial company. This issue raises data privacy and confidentiality concerns, as sensitive or proprietary code could be exposed to unintended parties. This could potentially lead to data breaches and intellectual property concerns [10]. LLMs can be trained on local computing units such as GPUs or private cloud architecture, e.g. TPUs. However, training these models requires expensive infrastructure and Meta's research team, for example, used over 3 million GPU training hours (about 378 years) to develop Llama 2 [11]. This resulted in a carbon footprint of 539 tCO_2eq, equivalent to what 52 people consume in one year in Germany [12]. Therefore, it is impractical for small companies or universities to train an LLM from the ground up.

However, natural language processing (NLP) has significantly progressed in recent years. Two methods have been developed to improve LLM response quality on repository-level programming in a more specific, cost-effective and privacy-focused manner. One promising solution is Retrieval-Augmented Generation (RAG) [13], which incorporates external data into the generative process. The data is stored as vector representation in a vector database and passed as context to the LLMs. These RAG pipelines improve the model's ability to deliver accurate and relevant responses and offer greater flexibility in terms of changes in the data source, low query latency [14, 15], and less hallucination in conversations [16]. Researchers employed an open-source model connected to a RAG pipeline capable of code generation and summarising [17]. Similarly, other studies have utilized a RAG pipeline for their repository-level models, attempting to extract additional information from the documentation to enhance the results further [18, 19].

Another way to increase the performance of the models is to fine-tune them with synthetic self-generated data. Thanks to parameter-efficient fine-tuning (PEFT) [20], models can now be fine-tuned locally and cost-effectively while ensuring privacy by saving the model weights locally. Using PEFT existing open source models such as Llama 2 [11], StarCoder [8], Open Assistant [21], or Mistral [22] can be trained as coding assistant. By using techniques such as prompt tuning [23, 24], low-rank adaptation mechanisms [25], or an adapter with a zero-init attention mechanism [20, 26], it is possible to fine-tune only subsets of weights. The resulting task- and domain-specific models, such as those for medicine [27] or, e.g., arithmetic tasks [28] surpass current state-of-the-art (SOTA) models, such as GPT-4.

To overcome the issue of generating a well-curated dataset by humans, Li et al. [29] introduced a self-alignment procedure enabling an LLM to generate training data through Self-Augmentation and Self-Curation. Self-Augmentation creates a training dataset by combining initial and unlabelled data to fine-tune the model on the repository source code using Supervised-Finetuning (SFT) [30] with QLoRA [31]. Additionally, Self-Curation evaluates the training dataset to filter out the best training examples. That procedure, called the Self-Alignment pipeline, helps the model better understand the repository's general purpose, file structure, and code style and enables the model to become a better coding assistant.

This thesis proposes a cost-effective and privacy-friendly approach to improve the performance of coding assistants on a specific repository by combining an RAG pipeline with a fine-tuned model trained on a self-augmented dataset.

1.1 Research Questions

Coding assistants can significantly improve developer's productivity [32]. Promising results from controlled experiments show that AI pair programming will soon be a reality [33], so further research on the repository level can improve this progress. A lot of research focuses on improving the general coding abilities of the models by training them on large corpora of source code data. However, this thesis focuses on combining repository source code and LLM to build an expert on this repository. Lewis et al. [13] showed that RAG pipelines empower LLMs with knowledge and, through the development of PEFT [20] models can now efficiently instruction fine-tuned with datasets on local infrastructure. Hence, this thesis compares coding assistants, e.g., LLMs, empowered by these two approaches.

For this investigation, the open-source model Mistral 7B (Instruct v0.2)[22] is used as a base model, connected to RAG pipelines, and fine-tuned. Mistral 7B is a pretrained LLM that outperforms Llama 2 7B, 13B [1] and CodeLlama 7B [1] on almost every benchmark and is, therefore, one of the best open-source models that can generate natural language and code at the same time.

Selecting the appropriate Python repository is crucial for the success of this thesis, as the repository should represent common repository structures and needs to be big enough to generate training data. Therefore, the CodeSearchNet corpus was analyzed [34]. It contains over 6 million programming functions with complete documentation in six programming languages from 13590 repositories. The Spyder IDE repository[8] is selected due to its abundance of short functions and documentation.

For the fine-tuning approach, the training dataset is generated by Self-Augmentation and Self-Curation techniques. On the other hand, for the RAG pipeline approach,

[8] https://github.com/spyder-ide/spyder/tree/master

the model uses a vector database to answer the research question. The source code is embedded using Instructor Base [35] and stored in a vector database.

Regarding the evaluation of the models, current research focuses either on benchmarks, e.g. HumanEval [5] or MBPP [36] or LLM-as-a-Judge [37]. LLM-as-a-Judge [37] is a powerful method that leverages a superior model to judge other models' responses. This method generates reliable results with Q&A pairs generated by humans or algorithms based on the repository.

This thesis presents a newly created dataset named SpyderCodeQA. It includes 325 question-and-answer pairs (Q&A pairs) from three question categories. To differentiate the model's performance, the dataset measures source code semantics understanding (140 pairs), dependency understanding (135 pairs), and knowledge of meta-information about the repository (50 pairs). Benchmarks are used to test the response quality of the presented models and demonstrate that the model's general coding abilities are consistent. LLM-as-a-Judge is used to test whether adding information through fine-tuning or RAG pipeline improves the response quality. This setup results in the following three RQs that are addressed in this master thesis:

RQ1: *How does fine-tuning an LLM with self-augmented data improve the performance of repository-level code question answering?*

RQ2: *How does a RAG pipeline improve the performance of repository-level code question answering?*

RQ3: *Does combining a fine-tuned model and an RAG pipeline show an interaction effect on the performance of repository-level code question answering?*

1.2 Structure of the thesis

This section gives an overview of the structure of the whole thesis. The thesis is divided into seven chapters, each focusing on a specific aspect. Starting with the Introduction in Sec. 1, this chapter introduces the topic and outlines the research questions and objectives of the thesis. The second chapter, Theoretical Foundation in Sec. 2, delves into the theoretical background of the research. It provides an in-depth review of language models 2.1, including attention mechanisms 2.2 and transformer architectures 2.2. Additionally, it discusses the training of large language models 2.3, including pre-training 2.3.1, fine-tuning 2.3.2, and PEFT. The chapter ends with an introduction to the concept of RAG and its applications 2.4. The third chapter, Related Work 3, comprehensively reviews existing research in the field. It discusses related work on repository-level programming 3.1, data augmentation 3.2, question-answering datasets 3.3, and the evaluation of LLMs 3.4. The fourth chapter, Evaluation Dataset 4, presents the dataset SpyderCodeQA used for evaluation. It describes the creation of the dataset, including the source code semantic dimension 4.1, dependencies dimension 4.2, and meta-information dimension 4.3. The chapter also analyses the dataset, including typical examples, token and sentence analysis, and question keyword diversity 4.4. The fifth chapter, Method 5, outlines the research methodology used in this thesis. It describes the data preprocessing steps 5.1, model details 5.2, self-alignment approach 5.3, retrieval-augmented generation pipeline 5.4, and the use of LLM-as-a-judge 5.5. The sixth chapter, Experiments 6, presents the experimental results of the thesis. It discusses the metrics used to evaluate the models, including LLM-as-a-judge and benchmarks 6.1. The chapter also presents the results of the self-alignment and RAG pipeline experiments, including the results on SpyderCodeQA and benchmarks. In the final chapter, Discussion 7 interprets the results of the thesis and answers the research questions. It discusses the implications of the findings and provides conclusions and recommendations for future research.

2. Theoretical Foundation

This chapter gives an overview of the theoretical foundation of this thesis. First, an introduction to language models is given in Sec. 2.1, explaining the development from n-grams to Transformer architectures. Additional information on the architecture of transformers and the attention mechanism is provided in Sec. 2.2. This is followed by an explanation of the training of LLMs in Sec. 2.3, including pre-training in Sec. 2.3.1 and Fine-tuning in Sec. 2.3.2. Finally, the concept of RAG is introduced in Sec. 2.4 and a discussion of embedding models in Sec. 2.4.1.

2.1 Language Models

In this thesis, LLMs are utilized to answer the research questions. Nowadays, they are predominantly based on the transformer architecture proposed by Vaswani et al. [38], leading to a significant breakthrough in NLP. Over the last decade, NLP has undergone a notable transformation, and this chapter provides an overview of the progress in this research discipline.

Language models, which aim to predict the next word in a sequence of text, have evolved significantly from simple statistical models to complex neural network-based architectures. The concept of language models dates back to the 1940s, when Claude Shannon, the father of information theory, introduced the idea of modelling language as a probabilistic process [39]. Early language models were based on statistical techniques, such as n-gram models, which predicted the next word in a sequence based on the frequency of word co-occurrences. These simple but effective models formed the basis for more complex language models.

In the 1980s, the introduction of Recurrent Neural Networks (RNNs) revolutionized the field of language modelling [40]. RNNs, designed to handle sequential data, could capture long-range dependencies in language, a limitation of traditional statistical models. Early RNN-based language models, such as the one proposed by Bengio, Ducharme, and Vincent [41], demonstrated improved performance over statistical models. However, RNNs suffered from the vanishing gradient problem. The vanishing gradient problem occurs when the gradients of the loss function concerning the weights in the early layers of a deep neural network become extremely small during backpropagation. This makes it difficult for the network to learn long-term dependencies, as the updates to the weights in the early layers become very small. Thus, the network effectively prevents itself from learning meaningful representations of the input data. The introduction of Long Short-Term Memory (LSTM) networks, a variant of RNNs, addressed the vanishing gradient problem [42]. LSTMs, with their ability to learn long-term dependencies, became common in NLP. LSTM-based language models, such as the one proposed by Graves [43], achieved SOTA performance on various language modelling tasks.

In 2013, a significant breakthrough in NLP occurred with the introduction of Word2Vec, a word embedding model proposed by Mikolov et al. [44]. Although dense vector word representations had been in use since 2003 [41], Word2Vec's efficient training procedure and large-scale training on unstructured text data allowed for the capture of complex relationships between words, such as gender, verb form and country-capital relationships. One year later, Sutskever, Vinyals, and Le [45] introduced sequence-to-sequence (seq2seq) learning, a general end-to-end approach for mapping one sequence to another using a neural network. This method, which employs an encoder and a decoder, has been widely adopted in NLP, particularly machine translation. The progress in seq2seq learning was so significant that Google replaced its traditional machine translation models with a neural seq2seq model in 2016 [46]. In a typical seq2seq architecture, the encoder model, such as an LSTM, processes the input token sequence to produce a continuous intermediate representation that captures essential information from the input sequence. The decoder model then processes this intermediate representation to generate the output sequence one token at a time, using the hidden state from the previous step and the previously generated token to predict the next output token.

2.2 Transformer Architectures

Despite the success of encoder-decoder architectures in seq2seq tasks, their performance has been limited by using RNNs as building blocks. The sequential nature of RNNs causes their performance to degrade as the length of the input sequence increases. A significant breakthrough in addressing this limitation was the introduction of the attention mechanism [47]. This innovation enabled the decoder to selectively focus on specific parts of the input sequence rather than relying solely on the encoder's final hidden state. The attention mechanism has since been refined and extended, with self-attention emerging as a key component of transformer models. This chapter will delve into the evolution of transformer architectures, explaining the self-attention mechanism, the architecture of the original implementation, and the latest refinements.

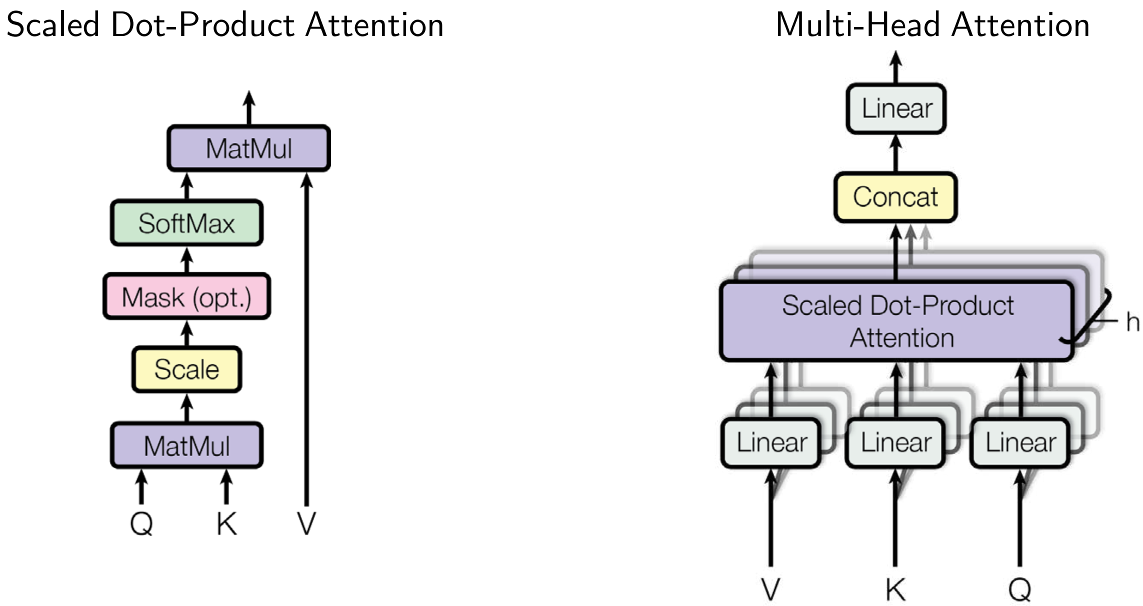

Figure 2.1: Overview of the Self-Attention mechanism and Multi-Head Attention proposed in the original implementation [38].

Self-Attention This section provides an overview of the formulas to understand the Self-Attention mechanism better and is based on the explanations of a blog post by Raschka, S.[1]. This mechanism is the core component of the Transformer architecture, allowing the model to capture dependencies between input elements and attend to different aspects of the input simultaneously. Self-attention is a mechanism that computes the representation of a sequence by relating different positions of the same sequence. Given an input sequence $X \in \mathbb{R}^{n \times d}$, where n is the sequence length, and d is the input dimension, the self-attention mechanism computes three matrices: the query matrix Q, the key matrix K, and the value matrix V. These matrices are obtained by linearly projecting the input X using learned weight matrices W^Q, W^K, and W^V, respectively:

$$Q = XW^Q, \quad W^Q \in \mathbb{R}^{d \times d_k} \tag{2.1}$$

$$K = XW^K, \quad W^K \in \mathbb{R}^{d \times d_k} \tag{2.2}$$

$$V = XW^V, \quad W^V \in \mathbb{R}^{d \times d_v} \tag{2.3}$$

[1] https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html

where d_k and d_v are the dimensions of the query/key and value vectors, respectively. As shown in Fig. 2.1 on the left, the attention scores are computed as the scaled dot-product between the query and key matrices, followed by a softmax function to obtain the attention weights:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.4}$$

The scaling factor $\sqrt{d_k}$ is used to prevent the dot-product from growing too large, which can cause the softmax function to have extremely small gradients.

Multi-Head Attention Multi-head attention is an extension of self-attention, allowing the model to attend to information from different representation subspaces jointly. Instead of performing a single attention function, multi-head attention linearly projects the queries, keys, and values h times with different learned projection matrices as shown in Fig. 2.1 on the right. Each of these projections is called an attention head.

For the i-th head, the query, key, and value matrices are computed as follows:

$$Q_i = XW_i^Q, \quad W_i^Q \in \mathbb{R}^{d_v \times d_k} \tag{2.5}$$

$$K_i = XW_i^K, \quad W_i^K \in \mathbb{R}^{d_v \times d_k} \tag{2.6}$$

$$V_i = XW_i^V, \quad W_i^V \in \mathbb{R}^{d_v \times d_v} \tag{2.7}$$

The attention function is then applied to each head in parallel:

$$head_i = Attention(Q_i, K_i, V_i) = softmax(\frac{Q_i K_i^T}{\sqrt{d_k}})V_i \tag{2.8}$$

The outputs of all attention heads are then concatenated and linearly projected to obtain the final output:

$$MultiHead(Q, K, V) = Concat(head_1, \ldots, head_h)W^O \tag{2.9}$$

where $W^O \in \mathbb{R}^{hd_v \times d}$ is a learned projection matrix.

The multi-head attention mechanism allows the model to capture different types of dependencies and relationships between elements in the input sequence, as each attention head can focus on different aspects of the input.

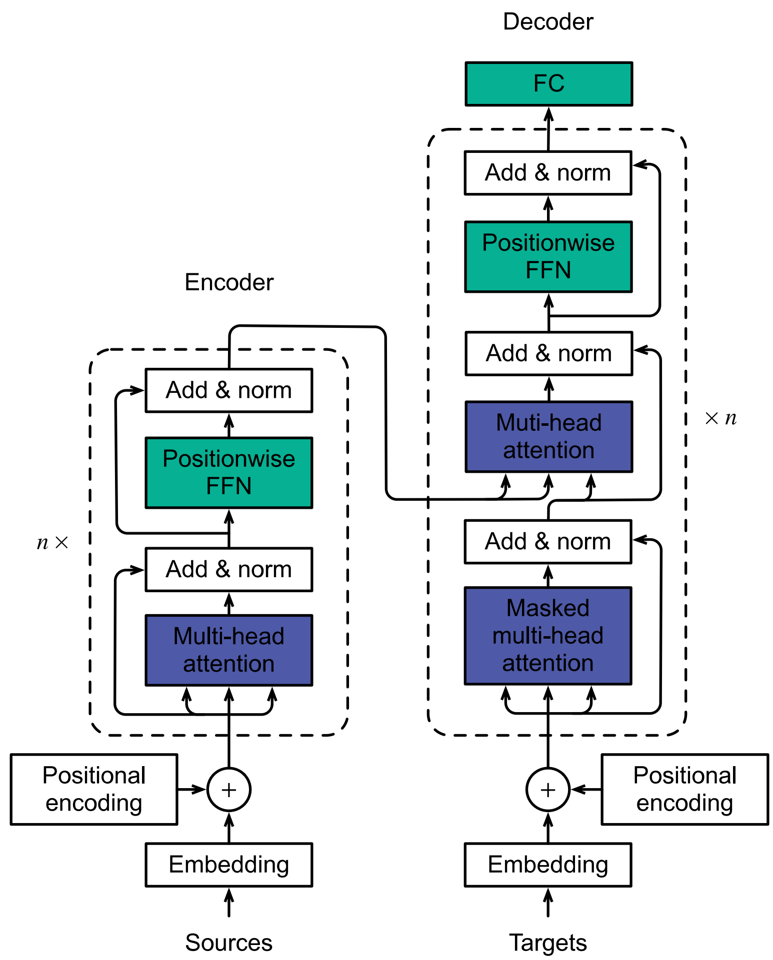

Figure 2.2: Overview of the original proposed Transformer Architecture[38][2]

Original Implementation The self-attention mechanism is the core component of the original transformer architecture, as illustrated in Fig. 2.2. It consists of an encoder-decoder structure and is primarily designed for seq2seq tasks, such as machine translation, where the input and output sequences are variable lengths [38]. Unlike traditional RNNs, the transformer relies solely on self-attention mechanisms to model relationships between input and output sequences. The encoder takes in a sequence of tokens (e.g., words or characters) and outputs a sequence of vectors (K, V, Q). The decoder then generates the output sequence, one token at a time, based on the encoder's output and the previous tokens generated.

[2] https://zahere.com/how-chatgpt-works-the-architectural-details-you-need-to-know

In the encoder, the input sequence is first embedded into a vector space using a learned embedding matrix. The embedded sequence is then fed into a stack of identical layers, each consisting of two sub-layers: a multi-head self-attention mechanism and a feed-forward network (FFN). The multi-head self-attention mechanism allows the model to attend to different parts of the input sequence simultaneously and weigh their importance. The output of the self-attention mechanism is fed into the FFN, which consists of two linear layers with a ReLU activation function in between. This FFN is used to transform the output of the self-attention mechanism.

In the decoder, the output of the encoder is fed into another stack of identical layers, each consisting of three sub-layers: a masked multi-head self-attention mechanism, a multi-head attention mechanism, and an FFN. The masked multi-head self-attention mechanism is similar to the one used in the encoder, except that it only attends to the previous tokens generated by the decoder. The encoder-decoder attention mechanism allows the decoder to attend to the encoder output. The encoder's output is used as the keys and values, while the decoder's output is used as the queries. The output of the encoder-decoder attention mechanism is then fed into an FFN, similar to the one used in the encoder.

Finally, the decoder's output is passed through a fully connected (FC) layer with a softmax activation function to generate the probability distribution over the output vocabulary.

Further Developments Since this revolutionary architecture, there have been several significant developments and improvements building upon the original architecture. One notable development was the introduction of the Bidirectional Encoder Representations from Transformers (BERT) [48]. BERT employed a multi-layer bidirectional Transformer encoder to generate contextualized word representations. The key innovation of BERT was using a pre-training objective called Masked Language Modeling (MLM), which involved randomly masking input tokens and training the model to predict the original tokens. This allowed the model to learn rich, context-sensitive representations that could be fine-tuned for various downstream NLP tasks, achieving state-of-the-art results.

Following BERT's success, several variants and improvements were proposed. RoBERTa [49] introduced modifications to the pre-training process, such as dynamic masking and larger batch sizes, leading to improved performance. DistilBERT [50] focused on creating a smaller, faster, and lighter version of BERT while retaining most of its performance. ALBERT [51] introduced parameter-reduction techniques to lower memory consumption and increase the training speed of BERT-like models.

Another line of development was the Generative Pre-training (GPT) series [52,

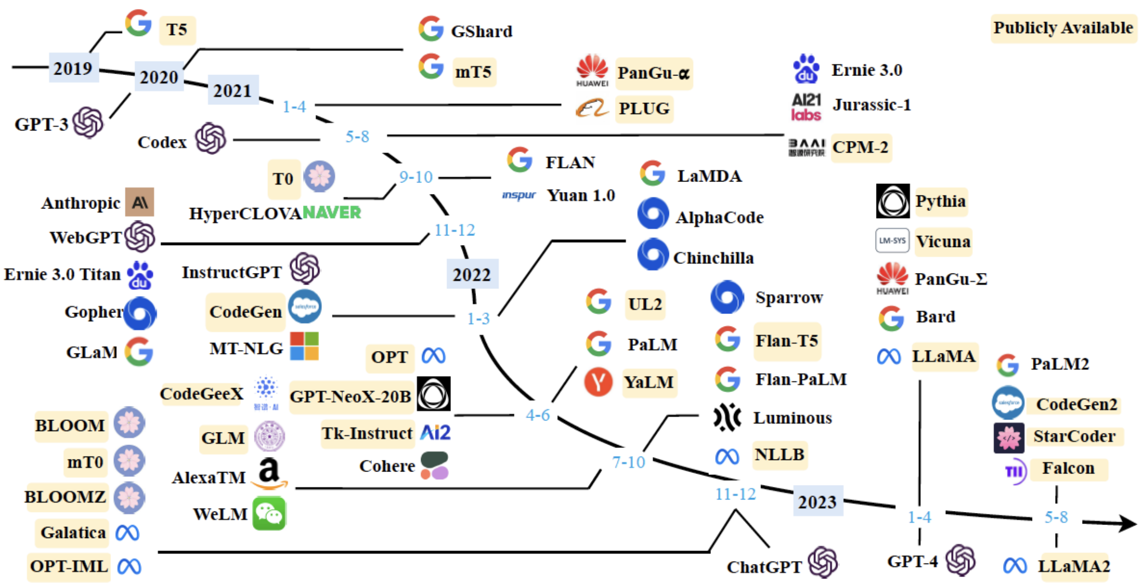

Figure 2.3: A timeline of existing large language models (having a size larger than 10B) in recent years. The timeline was established mainly according to the release date (e.g., the submission date to arXiv) of the technical paper for a model. LLMs with publicly available model checkpoints are in yellow colour. Extracted from Zhao et al. [55].

53, 54], which utilized a Transformer decoder for language generation tasks. GPT models were pre-trained using a Causal Language Modeling (CLM) objective, which involved predicting the next token in a sequence given the previous tokens. GPT-2 [53] and GPT-3 [54] further scaled up the model size and training data, demonstrating impressive language generation capabilities and paved the way to create ChatGPT, a GPT-based model that is trained for the conversation with human via text.

2.2.1 LLMs

Due to the ever-increasing number of parameters in the BERT and GPT-based models, the term Large Language Model (LLM) was created to describe this class of models. LLMs could generate accurate outputs but struggled with processing diverse inputs from users. However, this changed with the development of instruction models by Ouyang et al. [30], who trained them to interact with users using reinforcement learning from human feedback (RLHF). Consequently, the models became less toxic, less biased, and provided more accurate information when interacting with humans.

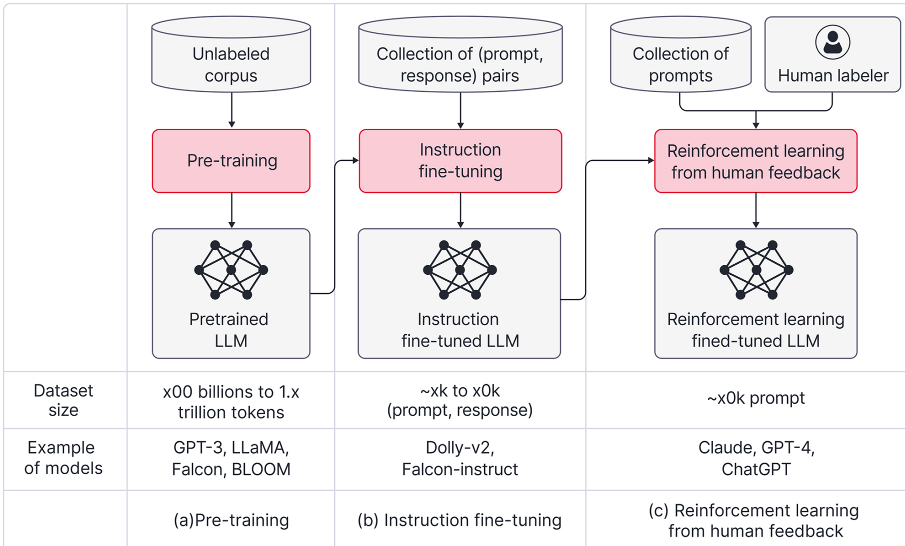

Figure 2.4: a) Pre-training involves using an unlabeled corpus containing hundreds of billions to a trillion tokens. b) Instruction fine-tuning is commonly applied after pre-training, using thousands to tens of thousands of (prompt, response) pairs as demonstration data for training. c) Reinforcement learning from human feedback (RLHF) is generally applied after instruction fine-tuning, using tens of thousands of prompts for training.[3]

OpenAI utilized this breakthrough to create ChatGPT - a conversational language model based on the GPT-3.5 architecture. ChatGPT generates human-like responses in various conversational contexts. Other models, like Llama [11] and Llama 2 [1], were published under an open-source licence and can be used for research.

According to a survey by Zhao et al. [55], the progress of LLMs has been rapid, making it difficult to stay updated on the latest research. A timeline of the existing LLMs is shown in Fig. 2.3, suggesting several companies working on developing LLMs. Additionally, companies such as OpenAI and Anthropic have not published their weights for fear of losing the lead to the competition, as the development of LLMs is now a billion- or trillion-dollar business. This effectiveness of LLMs in understanding and generating coherent and contextually relevant text has led to their exploration in new domains, such as medicine [27, 56] and law [57]. Despite their successes, LLMs present ethical concerns, biases, and interpretability issues that require addressing. This maximizes their potential in various applications, including automated evaluation systems.

2.3 Training LLMs

This section overviews the different training stages necessary to train LLMs. LLMs are usually trained using a combination of methods, each with a specific purpose in shaping the behaviour and performance of the model. Fig. 2.4 gives an overview of the training process, which can be categorized into three stages: pre-training, Supervised instruction fine-tuning, and reinforcement learning from human feedback (RLHF).

2.3.1 Pre-training LLMs

As illustrated in Fig. 2.4, pre-training is a crucial stage in developing LLMs. It involves training LLMs on unlabeled data corpus, often unsupervised, before fine-tuning them for specific tasks. This initial training phase provides the models with a general understanding of language structure and usage, which serves as a foundation for more specialized learning in subsequent stages. The methods used for pre-training vary depending on the architecture of the LLM.

For encoder-only models, such as BERT [48], pre-training typically involves masked language modelling (MLM) and next-sentence prediction (NSP). In MLM, some words in the input are masked (usually 15%), and the model is trained to predict these masked words based on the context. NSP, on the other hand, teaches the model to understand the relationship between sentences by predicting whether two given sentences are consecutive in the original text.

Decoder-only models, like the GPT-Family [52, 53, 54], Llama-Family [1, 11] or Mistral [22], use a probabilistic approach called causal language modelling (CLM). In CLM, the model is trained to estimate the probability distribution of a sequence of tokens $x = (x_1, x_2, ..., x_n)$ by maximizing the likelihood of the sequence given the previous tokens:

$$p(x) = \prod_{i=1}^{n} p(x_i|x_1, ..., x_{i-1})$$

where x_i is the i-th token in the sequence, and $p(x_i|x_1, ..., x_{i-1})$ is the conditional probability of the i-th token given the previous tokens. The model learns to predict the next token in the sequence based on the context provided by the previous tokens.

[3] https://is.gd/nDDzJ5

The objective is to maximize the log-likelihood of the training data:

$$\mathcal{L} = \sum_{x \in \mathcal{D}} \log p(x)$$

where $\mathcal{D}$ is the training dataset, and $p(x)$ is the probability of the sequence x according to the model. The autoregressive nature of CLM allows the model to generate coherent and contextually relevant text.

Encoder-decoder models, such as T5 [58] and BART [2], employ a combination of both encoder-only and decoder-only pre-training methods. During pre-training, the encoder is trained to map input sequences to a continuous representation, while the decoder is trained to generate output sequences based on this representation. This is often achieved through denoising autoencoding, where the input sequence is corrupted, and the model is trained to reconstruct the original sequence. Additionally, encoder-decoder models can be trained using a combination of MLM and CLM objectives, allowing them to leverage the strengths of both approaches.

2.3.2 Fine-tuning LLMs

Fine-tuning is the process of adapting a pre-trained LLM to a specific task or dataset, such as sentiment analysis, question-answering, or text classification, as shown in Fig. 2.4. During fine-tuning, the model's parameters are adjusted to optimize performance on the target task while leveraging the knowledge acquired during pre-training. There are two main approaches to fine-tuning LLMs: full fine-tuning and parameter-efficient fine-tuning (PEFT) [20]. Full fine-tuning involves updating all the model weights, including the attention and output layers, during training. This approach allows the model to learn to deal with a particular subject but requires significant computational resources and a well-curated dataset. On the other hand, PEFT focuses on training only a subset of weights, thereby reducing costs and shortening training time.

Supervised Fine-Tuning Supervised fine-tuning (SFT) is a common method that requires sets of labelled input-output pairs. The objective function for SFT is typically measured using a loss function, such as cross-entropy loss or mean squared error. For classification tasks, the cross-entropy loss is commonly used:

$$\mathcal{L} = -\sum_{i=1}^{N} y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$$

where N is the number of classes, y_i is the true label, and $\hat{y}_i$ is the predicted probability. For regression tasks, the mean squared error (MSE) is often used:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where y_i is the true value, and $\hat{y}_i$ is the predicted value. The optimization process for SFT involves updating the model's parameters to minimize the loss function. This is typically done using an optimizer, such as stochastic gradient descent (SGD), Adam, or RMSProp. The optimizer iteratively updates the model's parameters based on the gradients of the loss function with respect to the parameters.

One of the most popular SFT methods is instruction tuning [59, 60, 30, 61], which has emerged as one of the most powerful methods to improve model performance. With instruction tuning, the input is a natural language task description, and the output is an example of the desired behaviour. This approach enables the model to learn to understand and follow instructions, allowing it to generalize well to unseen tasks and datasets. Instruction tuning is particularly effective in tasks that require complex reasoning, such as question-answering, text classification, and dialogue generation. By providing the model with explicit instructions, instruction tuning allows the model to focus on the specific task at hand rather than relying on implicit patterns in the data. Additionally, instruction tuning can be used with other SFT methods, such as few-shot learning and transfer learning, to improve model performance.

Parameter-Efficient Fine-Tuning (PEFT) PEFT techniques have emerged as a promising approach to optimize the fine-tuning process of LLMs by selectively updating a subset of parameters, thereby reducing the computational cost and required resources [26]. These techniques focus on learning a small number of parameters for the task at hand by designing additional layers [25], adding prepending additional tokens [24, 20], or decomposing weight gradients into specific matrices [25]. One of the representative cutting-edge PEFT techniques is Low-Rank Adaptation of LLMs (LoRA) [25], which freezes the model weights and injects low-rank trainable matrices into the attention layers of the Transformer architecture [38], reducing the number of trainable parameters.

Additionally, Prompt tuning [24] and Prefix tuning [62] are suitable techniques, which involve prepending virtual tokens to the input tokens of the LLM or inserting virtual tokens in all the layers of the target model, respectively. These virtual tokens are differentiable, allowing them to be learned through backpropagation during fine-tuning while the rest of the LLM remains frozen. Furthermore, QLoRA [31] combines

LoRA with model quantization, enabling the fine-tuning of LLMs with less GPU memory by reducing the precision of floating point data types within the model. By updating a few parameters, PEFT methods aim to optimize model performance while reducing the required resources and tuning time, making them particularly suitable for knowledge editing, which requires efficient modification of model behaviour.

2.3.3 Reinforcement Learning from Human Feedback (RLHF)

Reinforcement Learning from Human Feedback (RLHF) is a machine learning paradigm that combines elements of reinforcement learning and supervised learning to enable AI systems to learn and make decisions in a more human-aligned manner [63]. RLHF introduces human feedback as a valuable source of guidance, which can help AI systems navigate complex decision spaces, align with human values, and make more informed and ethical choices [64].

Fig. 2.4 illustrates the RLHF process, which consists of several key steps. Firstly, prompts are collected to train a reward model through human labelling. Subsequently, the language model is fine-tuned using reinforcement learning techniques, followed by deploying and iterating the model [65]. During the fine-tuning process, the reward model guides the model's actions, and the agent aims to maximize cumulative rewards based on the reward model's predictions.

2.4 Retrieval-Augmented Generation (RAG)

The concept of RAG was introduced by Lewis et al. [13] in 2020, although similar ideas were also proposed by Guu et al. [67] with Retrieval-Augmented Language Model (REALM). The goal of RAG is to combine the strengths of parametric memory (e.g., large language models) with non-parametric memory (e.g., databases) to enhance model capabilities and reduce hallucinations. Figure 2.5 shows a SOTA implementation of a naive RAG pipeline, as described in [66]. This pipeline consists of three primary steps.

Firstly, the indexing step (illustrated in the top right corner) involves splitting documents into chunks, encoding these chunks into dense vectors using a neural network, and storing them in a vector database. This allows for efficient similarity-based retrieval of relevant documents. The second step is retrieval (illustrated in the bottom right corner), where the top k chunks of documents most semantically similar to the input question are retrieved from the vector database. This is typically done using a similarity metric such as cosine similarity or dot product. Finally, in

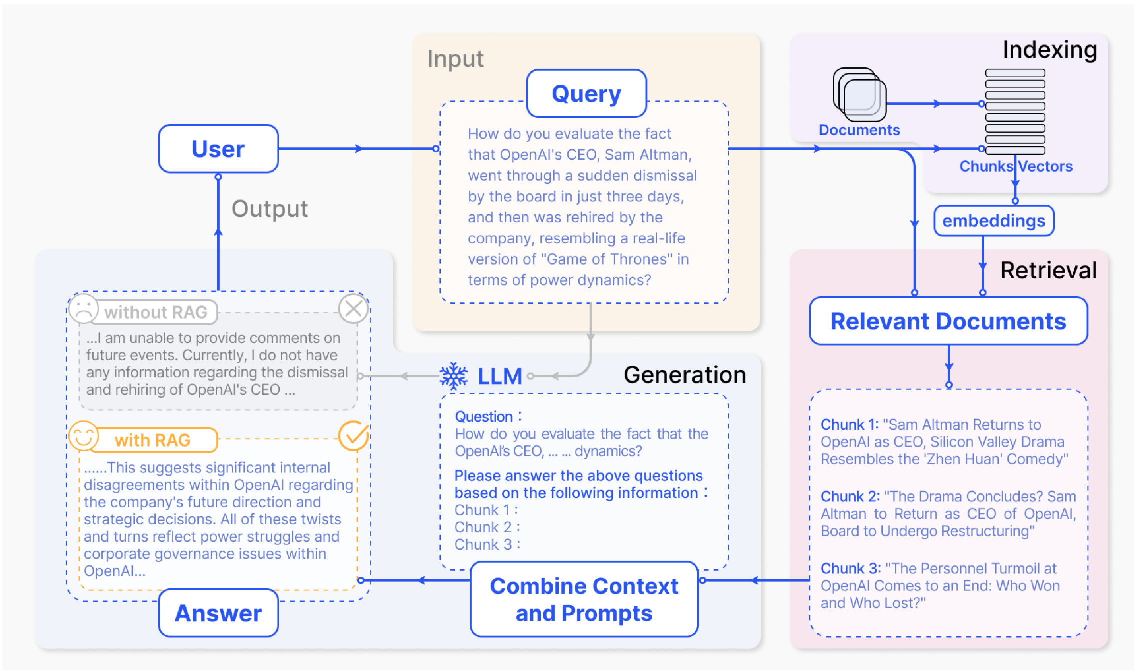

Figure 2.5: A representative instance of the RAG process applied to question answering. It mainly consists of 3 steps. 1) Indexing. Documents are split into chunks, encoded into vectors, and stored in a vector database. 2) Retrieval. Retrieve the Top k chunks most relevant to the question based on semantic similarity. 3) Generation. Input the original question and the retrieved chunks together into LLM to generate the final answer. Extracted from Gao et al. [66].

the generation step (illustrated on the left), the original question and the retrieved chunks are input into an LLM to generate a coherent and accurate final answer. The LLM leverages the context and information provided by the retrieved chunks to produce a response that is more informed and relevant than what would be possible with a single pass through the input question alone.

Several research papers have integrated RAG pipelines with LLMs, such as for law [57] or medicine [56], to ensure domain knowledge. The advantage of RAG pipelines is that they are less expensive to create than training LLMs and can better accommodate updates. However, challenges in RAG pipelines include suitable information representation, suitable algorithms to find relevant information, and selection based on ranking methods. Additionally, the problem of code retrieval [68] requires the combination of natural language with source code in vector representation.

2.4.1 Embedding Models

In modern RAG pipelines, the choice of embedding model plays a crucial role in retrieving relevant documents from the database. All the more so in this work, the combination of natural language and code represents a further challenge. However, recent advancements in dense embedding models have significantly improved the efficiency and effectiveness of RAG systems.

Embedding models are typically encoder-only models or, more recently, decoder LLMs with a special embedding head on top. These models take input data, such as text or code, and output a dense vector representation that captures the semantics of the input. This contrasts language models, which output token IDs or text. The output vector from an embedding model can then be used to calculate the similarity between different inputs, enabling efficient retrieval of relevant documents.

As discussed in Gao et al. [66], the choice of embedding model is crucial in RAG pipelines, where retrieval is achieved by calculating the similarity between question and document chunk embeddings. Recent research has introduced prominent embedding models, such as AngIE [69], Voyage[4], and BGE [5], which benefit from multi-task instruction tuning.

In selecting an embedding model, there is no one-size-fits-all answer. However, some models are better suited for particular use cases. For instance, mix/hybrid retrieval approaches can leverage complementary relevance information from sparse and dense embedding models. Fine-tuning embedding models on domain-specific datasets can also mitigate discrepancies between pre-training corpora and target domains, particularly in specialized fields like healthcare and law. Additionally, fine-tuning can align retrievers and generators using techniques like LSR (LM-supervised Retriever) and PROMPTAGATOR [70].

One promising generalistic model that reached SOTA in over 70 tasks is INSTRUCTOR [35]. Introduced as a novel method for computing embeddings, which can be customized for various downstream tasks and domains that come in three different model sizes (100M, 300M, 1.5B parameters). Unlike specialized encoders, INSTRUCTOR is a single embedder capable of generating text embeddings without additional training, provided with task instructions that outline the specific use case. Additionally, INSTRUCTOR can retrieve code, making it suitable for this thesis.

[4] https://docs.voyageai.com/embeddings/

[5] https://github.com/FlagOpen/FlagEmbedding

3. Related Work

This chapter provides an overview of the existing research related to this thesis. The field of coding-specific tasks has seen significant advancements in recent years, with the development of task-specific LLMs for general coding tasks and repository-level programming discussed in Sec. 3.1. Building upon this foundation, researchers have explored the application of data augmentation techniques to improve the performance of LLMs, as described in Sec. 3.2. Code-based question answering has also been a focus of research, with the development of datasets and benchmarks to evaluate the performance of LLMs, as discussed in Sec. 3.3. Finally, evaluating LLMs is a crucial aspect of this field, with various approaches and benchmarks being explored, as described in Sec. 3.4.

3.1 Repository-level Programming

Developing coding assistants for repository-level programming builds upon a rich research foundation in coding-specific tasks. Task-specific LLMs have been developed for a variety of tasks, including summarizing text, generating questions [2], and even performing coding-specific tasks such as coding summarization [3], coding generation [4, 5, 8, 71], answering questions regarding source code [6], and generating code documentation on repository-level [7]. Also, research on building interfaces for coding assistants was conducted in the near past [72].

Early studies [73] proposed using nested n-gram models to leverage the locality of directories within a project for code completion. This idea was further built upon by following works from Shrivastava, Larochelle, and Tarlow [74], which introduced a framework for generating example-specific prompts using context from the entire repository. Similarly, Zhang et al. [75] proposed RepoCoder, a framework that addresses the challenge of code completion at the repository level by incorporating useful information spread across different files. In more recent studies, the boundaries of coding assistants at the repository level have even been pushed further. Bairi et al. [76] framed repository-level coding as a planning problem and presented a task-agnostic framework called CodePlan to solve it. This framework synthesizes a multi-step chain of edits, where each step results in a call to an LLM on a code location with context derived from the entire repository, previous code changes, and task-specific instructions. Other related works have focused on specific aspects of repository-level coding assistants, such as generating prompts for few-shot learning [77] using RAG pipelines and effect debugging on a repository level [78].

Recent studies have explored the application of instruction fine-tuning with PEFT techniques for coding tasks. Weyssow et al. [79] demonstrated the effectiveness of PEFT for coding tasks on various models, highlighting the effectiveness of QLoRA for fine-tuning. They showed that PEFT can significantly improve the performance of coding-related tasks while minimizing the computational resources required for fine-tuning. In a related study, Yuan et al. [80] investigated the performance of instruction fine-tuned models on various coding tasks, including defect detection, clone detection, assertion generation, and code summarization. Their findings suggest that instruction fine-tuned models are equally effective as non-instruction models for these tasks. This implies that fine-tuning with SFT and additional information does not necessarily improve the model's skills in general coding tasks.

Researchers have also explored the combination of fine-tuning and RAG pipelines. Ovadia et al. [81] fine-tuned three different open-source models using unsupervised fine-tuning and injected additional information using the RAG pipeline. Their results showed that while unsupervised learning struggles to consistently inject information into the LLM, the RAG pipeline consistently outperforms the baselines. This suggests that the RAG pipeline is an effective approach for incorporating external knowledge into LLMs. Building upon these findings, Soudani, Kanoulas, and Hasibi [82] investigated the interaction between SFT and RAG pipelines. Their research suggests that SFT and RAG pipelines can benefit from each other, leading to improved performance on coding tasks. By combining the strengths of both approaches, models can leverage the knowledge gained from instruction fine-tuning while incorporating relevant information from external sources through the RAG pipeline.

Overall, these studies show the potential of PEFT, SFT and RAG pipelines to improve the performance of LLMs in coding tasks. Combining these techniques can lead to more effective and efficient coding assistants handling various coding-related tasks while minimizing the computational resources required for fine-tuning. This thesis combines all these findings by using a self-generated dataset as a training dataset and using SFT with QloRa to train it effectively and time-efficiently on Mistral. Additionally, it tries to investigate further the interaction effect that Soudani, Kanoulas, and Hasibi [82] investigated.

3.2 Data Augmentation

Collecting datasets can be a tedious and expensive task that requires expertise and is time-consuming. Although datasets already exist for most tasks in machine learning, they often lack size and variety and new data points must be created or augmented by the existing data points. To automate this process for LLMs, Wang et al. [59] introduced *Self-Instruct*. This approach uses ChatGPT [54] in-context learning ability to generate many instructions from a pre-defined set of human-annotated instructions that cover various topics and task types used for fine-tuning. The generated instructions are then filtered to ensure their quality, and the process continues until the desired data volume is achieved. Several studies followed up on this system as referred in the survey of Ding et al. [83], resulting in the creation of Alpaca [61], a fine-tuned variant of Llama 7B that is trained on synthetic data generated by ChatGPT. Regarding other domains, Cui et al. [57] used this approach to train Llama on augmented law data samples, and Peng et al. [84] showed that augmented data can improve machine translation from Chinese and English. Li et al. [29] introduced a novel approach to data augmentation called Self-Alignment. This method uses a small set of high-quality seed data to create a foundation of well-designed data samples. These seed samples serve as a starting point for generating additional data points. After generating new samples, the Self-Alignment approach employs a Self-Curation step, where the generated samples are rated and filtered based on their quality and relevance to the target task. This ensures that only the most informative and valuable samples are retained for further use in training or fine-tuning the model.

In source code learning, Dong et al. [85] explored various data augmentation techniques for tasks such as Code Refactoring, Problem Classification, and Bug/Clone Detection. They employed different approaches to transform the source code, representing it as either text or graphs. The transformed code was then used to train models such as Feed-forward Neural Networks (FNN), Convolutional Neural Networks (CNN), and Graph Convolutional Networks (GCN) on the respective tasks. Although the study yielded mixed results regarding the effectiveness of data augmentation in coding tasks, it is crucial to consider that the models used in this research are no longer considered state-of-the-art. Focusing more on LLMs, Zhang et al. [86] used data augmentation to annotate an existing dataset with custom function calls that invoke an API call to get additional information about the problem. Another approach was used by Patil et al. [87]. They manually created a well-curated dataset of API calls and used *Self-Instruct* to generate instruction-API pairs by GPT-4. These studies highlight how data augmentation can be applied to source code learning and LLMs. While the effectiveness of data augmentation may vary depending on the specific task and model architecture, these approaches offer promising avenues for improving the performance of coding-related models by enriching the training data with additional information and diverse examples.

3.3 Q&A Datasets

Code-based question answering is a subfield that focuses on responding to code-related queries. Unlike generative approaches, retrieval-based code Q&A aims to find the most relevant code snippets from a large code corpus to satisfy user requests. To evaluate the performance of current models, Husain et al. [34] introduced CodeSearchNet, a collection of datasets and benchmarks created by mining large-scale comment-code pairs from public GitHub repositories. Similarly, Liu and Wan [88] presented CodeQA, a free-form code question-answering dataset designed to assess the code comprehension capabilities of language models. CodeQA was constructed using existing code summarization datasets mined from GitHub, focusing on two popular programming languages: Python and Java. The dataset synthesizes various types of Q&A pairs by leveraging code comments and documentation strings, employing manually curated rules, templates, and a variety of NLP toolkits.

Recent work has been focused on constructing code Q&A datasets from real-world scenarios. CoSQA [89] mines real-world user queries from Bing search logs that were labelled if the provided answer is the solution to the question. Educational programming Q&A datasets have also gained attention. CS1QA [90] collects student questions and answers from teaching assistants on an online forum designed for an introductory Python programming course. This dataset offers insights into the educational applications of code-based question answering. Another recently released Q&A dataset by Li et al. [91] is focused on mixed-modal data that combines code and text. This dataset provides a more realistic simulation of the interaction between humans and a coding assistant. The dataset was created to train models to perform retrieval-based code Q&A, which involves finding the best possible code.

Although these Q&A datasets help measure the interaction of models and humans, they are unsuitable for repository-level programming tasks for various reasons. Unfortunately, CodeSearchNet [34] only consists of pairs of code and commits and is, therefore, difficult as there is no direct question-answer interaction. The same applies to CodeQA [88], which used the documentation and the comments to match natural language and code. While CoSQA [89] consists of actual human queries, the queries are only related to general coding tasks and have no label for a repository, which makes it challenging to use the Q&A pairs as training data to measure the performance of a specific repository. CS1QA [90] was not suitable because most of the questions were related to student's homework, which was not publicly available. Unfortunately, the most promising dataset, ProcQA [91], was not published until the thesis was completed. It was, therefore, decided to create a separate evaluation dataset that measures performance on a specific repository. All details for the creation are described in Chap. 4.

3.4 Evaluation of LLMs

Evaluating the capabilities of LLMs has been challenging due to their vast and diverse abilities and the lack of standardized benchmarks to measure human preferences in this rapidly evolving field. While traditional metrics such as BLEU [92] and ROUGE [93] are commonly used to assess deep neural networks in NLP, they may not accurately capture the complex abilities of LLMs. As highlighted in a recent survey [94], there is a pressing need to develop evaluation methods that can measure the reasoning, robustness, ethical considerations, and other capabilities of LLM outcomes. In response, much research has focused on developing novel evaluation approaches for LLMs. These approaches can be broadly categorized into two types: benchmarks that assess general skills, such as reasoning and harmlessness, and those that evaluate specific skills, such as coding and model-based evaluation, which use superior models as judges to assess model performance.

Benchmarks Current SOTA papers use benchmarks, such as HumanEval [5], MBPP [36], HumanEval-X [95], DS-1000 [96] to measure the performance of LLMs. Core Knowledge benchmarks check the core capabilities of pre-trained LLMs using zero-shot and few-shot settings. Typically, LLMs need to generate a short, specific answer to benchmark questions that can be automatically validated [37]. Examples of Core Knowledge Benchmarks include measuring Reasoning Abilities with ARC [97] or Meaning to Comprehend with HellaSwag [98]. Measuring specific skills such as programming skills is achieved using HumanEval [5], HumanEval-X [95], MBPP [36] or DS-1000 [96]. In this case, the model has to generate source code for a problem, and unit tests are used to verify the functional correctness of the code.

LLM-as-a-Judge In contrast to Benchmarks that are used to measure the performance of models based on the functional correctness of code (Humaneval [5], MBPP [36]) or evaluate the performance using Humans [11], it is possible to use superior models to judge other models output. Model-based evaluation is used in various papers [84, 37, 99, 23, 100, 101] to measure the responses of LLMs.

In the context of automated evaluation systems, Zheng et al. [37] proposed three variations of Model-based-evaluation referred to as LLM-as-a-Judge. The first, pairwise comparison [84, 99], involves directly assessing two answers to determine superiority or a tie. The second, single answer grading, assigns a score directly to a response [100, 23]. The third, reference-guided grading, incorporates a reference solution, beneficial for math problems [99]. A similar approach was taken by Lin and Chen [102]. Unlike Zheng et al. [37], Lin and Chen [102] model rated on a scale from 0-5 or 0-100, and evaluated appropriateness, content, grammar, and relevance to distinguish the quality of the responses. Previous studies were also conducted to train a specialized evaluation model called PandaLM [103] to further enhance the model's judgement capabilities.

3.5 Coding Related LLMs

Coding-related LLMs, such as Codex [5], StarCoder [8], WizardLM [104], CodeGen2 [4], and CodeLlama [71], are designed to process and generate code in various programming languages. These models are trained on large datasets of code, allowing them to learn the syntax, semantics, and patterns of programming languages. As a result, they can assist developers in tasks such as code completion, code generation, and code review.

One key difference lies in their training objectives. While standard LLMs are typically trained to predict the next token in a sequence of text, coding-related LLMs are trained to optimize specific coding-related tasks, such as code generation, code correction, or code summarization. For example, WizardLM employs a hierarchical attention mechanism to capture long-range dependencies in code [104]. In contrast, CodeLlama utilizes a multi-task learning framework to jointly train on multiple coding-related tasks [71].

Another distinct characteristic is their ability to handle the structural complexity of code. Codex, for instance, uses a novel neural architecture that combines a transformer encoder with a syntax-aware decoder to generate syntactically correct code [5]. StarCoder, on the other hand, incorporates a graph-based neural network to model the structural relationships between code elements [8].

4. Evaluation Dataset

The thesis addresses the research question of how well coding assistants can perform at the repository-level programming. Therefore, a dataset was necessary to evaluate the performance of the models. It is rare to find information on code-based question answering, so a new evaluation dataset called SpyderCodeQA was created as part of this thesis. This chapter provides an overview of the process involved in its creation. The purpose of this dataset is to measure the model's ability to comprehend the fundamentals of a repository, including its structure, code, and purpose. To evaluate this ability, various sources such as books [105, 106], papers [107, 108], and blog articles[1][2] were examined to determine the essential components of Python repositories. Due to the diverse range of language applications, such as GUI-based programming, machine learning applications and game programming [108], it is difficult to define a typical structure of a Python repository.

Therefore, we will start with the smallest possible repository unit: one Python file containing the source code. If a model can understand the source code at this level, it must also comprehend its syntax and semantics. Hence, *source code semantics comprehension* is in this thesis defined as understanding the containing text and code elements about the repository's source code and being able to answer semantic questions about it.

The Python programming language is an organized architecture comprising individual files grouped as modules and packages [105, 106]. Depending on the selected architecture pattern (MVC, MVT, etc.), modules are structured individually, which makes it hard to analyze them. Nevertheless, these relationships between the files are essential for understanding the repository. Hence, dependency comprehension is defined in this thesis as understanding the relationships between files within the repository and between files and imported libraries.

The last source of information is general information about the repository. Prana et al. [108] showed that developers use README files for sharing various information such as build commands, requirements or legal information of the repository. Hence, meta-information comprehension is defined in this thesis as understanding all information about the repository unrelated to the source code.

This means that a coding assistant responsible for a specific coding repository should be able to answer questions on all three dimensions to guarantee comprehensive support, as shown in Fig. 4.1. Therefore, this thesis developed for each of the

[1] https://docs.python-guide.org/writing/structure/

[2] https://is.gd/8ylpcB

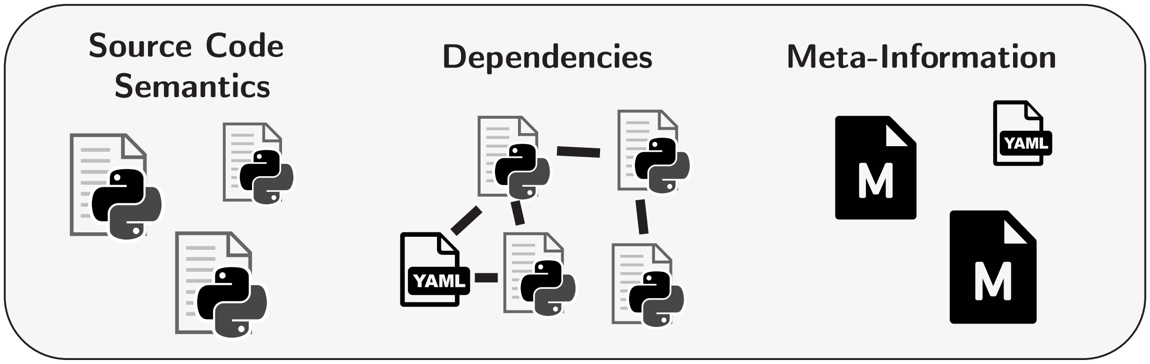

Figure 4.1: Overview of the three dimensions of the evaluation dataset. The dimensions include Source code semantics, Dependencies, and Meta-Information Q&A, each containing Q&A pairs that serve a specific purpose. These dimensions are designed to provide comprehensive information about the source code files, their relationships with modules and libraries, and general information about the repository.

dimensions a Q&A dataset to measure it. The creation process resulted in a dataset comprising a total of 325 samples. Table 4.1 shows the distribution of samples per dimension.

Dimension	**Code Semantic**	**Dependencies**	**Meta-Information**	**Total**
	140	135	50	**325**

Table 4.1: Number of Samples per Dimension

This chapter is divided into five sections to give an all-embracing overview of the creation process, statistics and its limitations for each dataset dimension. It includes details about the study conducted for the source code semantics comprehension dimension (Sec. 4.1), the creation of the dependency comprehension dimension (Sec. 4.2), and the meta-information Comprehension dimension (Sec. 4.3). After the creation process, a data analysis of the whole dataset follows, which explains various critical metrics in Sec. 4.4 and gives dataset examples for each dimension. Additionally, the challenges faced during data collection and the limitations of the dataset are described in Sec. 4.5.

4.1 Source Code Semantic Comprehension

For creating the source code semantics comprehension dimension, ten computer science experts manually created the Q&A pairs using the Spyder IDE repository source code[3]. An online study was conducted to create these pairs, and a custom web application was developed using Python Django as a backend service and HTML, CSS, and vanilla JavaScript with Bootstrap 5 for the user interface. The web app was hosted on a private home server during the data collection. Participants in the study were given two different tasks. Firstly, they had to create Q&A pairs for one of the 5673 snippets randomly selected from the open-source Python repository Spyder IDE. Secondly, they had to rate Q&A pairs from other participants to ensure the quality of the pairs on a 1-10 scale. Each participant was required to create ten Q&A pairs and rate ten pairs. Participants were also allowed to create and rate another set of ten Q&A pairs, resulting in twenty pairs.

4.1.1 Creation of Code Snippets

During the online study, participants were given a random code snippet from the open-source Python Spyder IDE code repository. These snippets were generated using the LangChain package's document loader and text splitter[1]. The 2083 Python files in the repository were divided into 5673 text chunks to create these code snippets. The source code was chunked using Python syntax and specific cutting points like `\nclass, \ndef`, and `\n\tdef`. Each chunk was not larger than 2000 characters. If the splitter within the chunk size found none of these cutting points, the splitter uses secondary cutting points such as `\n\n, \n` and `" "`. In addition to the source code, meta-information about the code snippets were stored. That included the name and module of the file and the start and end lines of the source code. The procedure for identifying the start and end line involved fetching the file path of the code snippet and comparing its content with the original content of the file. It then located the starting line of the snippet by matching its first line with the lines in the file and determined the end line based on the snippet's length. The function also accounted for edge cases where the snippet may not be found within the file or consists of only one line. After creating chunks of source code and meta-information, the data was stored in an SQLite database using Django object-relational mapping in Python.

[3] https://github.com/spyder-ide/spyder/tree/master

[1] https://python.langchain.com/docs/modules/data_connection/document_transformers/

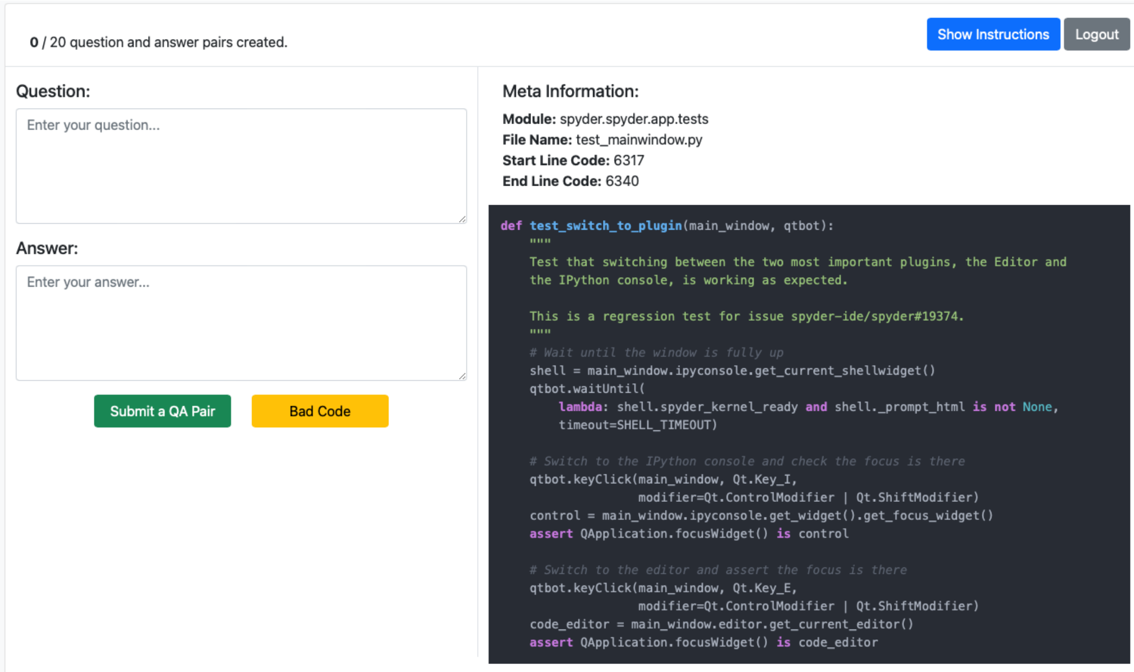

Figure 4.2: Web App frontend for creating Q&A pairs. Two input fields are on the left for entering questions and answers, and the Code snippet is on the right. Users submit a Q&A with the green button and mark it as Bad Code, e.g. code snippet is not understandable, with the yellow button.

4.1.2 Study Sample

Ten participants with a minimum master's degree in computer science-related fields participated in the online study. 90% of the participants had studied for more than ten semesters in total. The participants self-rated their general coding abilities at a mean of 3.8 on a scale of 1-5. Their average Python coding abilities and working experience were 4.3 and 3.5, respectively. The online study generated 190 Q&A pairs, aligning with other domain datasets like HumanEval ($N = 164$) [5]. One participant completed one iteration, creating and rating 10 Q&A pairs, while the other nine participants were able to create two iterations, resulting in a total of 20 Q&A pairs.

4.1.3 Execution of the Study

Participants were given login credentials via messenger or email with a link to the web application. The web app's homepage is shown in Fig. A.1. The left side of the page contained background information about the study, information about the repository, and some hints about the study process. The login panel was presented on the right side. Once logged in, each participant followed the same procedure.

Before executing the study, each user was asked to provide personal information. The required information included their highest computer science degree (Bachelor, Master, PhD, etc.), the number of semesters studied in total (rated on a scale of 1-10+), their self-rated coding skills (general and Python, rated on a scale of 1-5), and their field of study. A screenshot of the form to prompt users for this information is shown in Fig. A.2. This information was only collected to filter out bad Q&A pairs when participants had low coding or working experience.

Next, participants were given instructions and examples for the data creation task, which can be found in Appendix A.3 to A.9. This was done to give the participants an idea of what good Q&A pairs should look like and how they should behave if they do not understand something or if something is not understandable. The interface for the creation task is shown in Fig. 4.2. The left side of the interface contained two text areas, one for entering the question and the other for entering the answer. On the right side, the code snippets from the repository were displayed, along with meta information such as the module name, file name, and the start and end line of the code snippet.

The instructions for the rating task are given in Appendix A.10. Participants are informed that 0 represents a poor Q&A and 10 represents a perfect Q&A, along with the specific criterion for rating. Fig. 4.3 shows the process for rating the Q&A pairs. The interface for the rating task is similar to the creation task. However, the text areas have been replaced with rating forms. Participants could rate questions and answers separately on a scale of 1 to 10 and leave an optional comment.

Users could pause the study by logging out and resuming where they left off later, as the app automatically saved their progress. Upon completion, the participants saw an end screen (Appendix A.11) displaying the number of Q&A pairs they had created and rated. The execution duration of the study lasted an average (median) of 1 hour and 22 minutes, with the fastest participant finishing in 38 minutes and the slowest in 8 hours and 18 minutes. This large number is because the participants could interrupt the study to continue it later.

4.1.4 Data Aggregation

After the initial evaluations, any Q&A pairs receiving a rating below three were automatically removed from the dataset. For pairs with a rating between three and five, experienced annotators manually checked whether the pair was useful or could

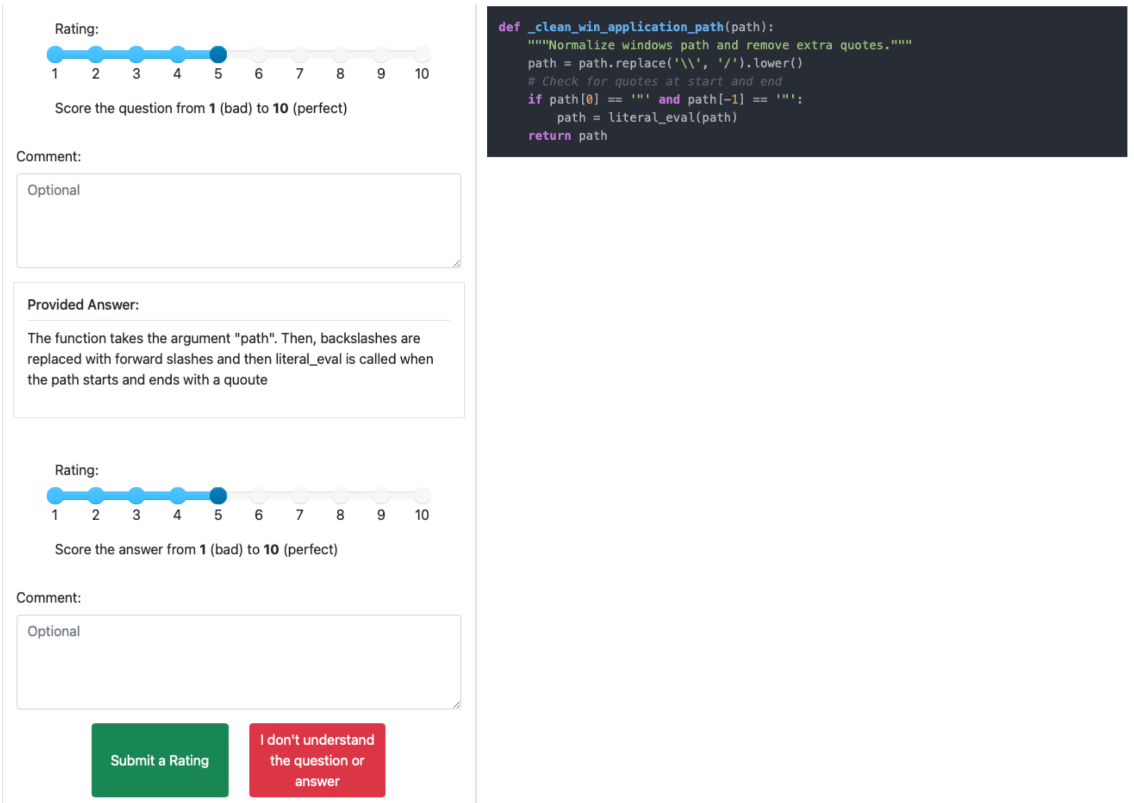

Figure 4.3: Web App frontend for rating Q&A pairs. Two slider inputs are on the left for entering a rating from 0 (bad) to 10 (perfect), and the Code snippet is on the right. Users submit a rating using the green button. Understanding problems with rating the Q&A pair resulted in submitting the red button.

be removed. The criteria used for the rating were functional correctness of code, understandable questions, whether the Q&A pairs were related to the repository and whether the question was detailed enough. As a result of this quality control process, the final dataset was shrunk and consisted of 140 Q&A pairs. This process ensured a strong and dependable foundation for the subsequent analysis.

4.2 Dependencies Comprehension

In this thesis, dependencies comprehension is defined as the ability to understand which files reference each other and which packages are required to execute the source code in the targeted file. Therefore, the source code of the Spyder IDE was analyzed. The goal was to create Q&A pairs for dependencies comprehension to measure the ability of the model to understand the dependencies between source code files. An algorithm was developed that uses abstract syntax trees (AST) to identify dependencies between files, modules, and libraries in a repository. This algorithm only recognizes 1-hop dependencies between two files, meaning connections across multiple files are not recognized. However, this algorithm uses the imports from the source code to identify which files, modules, and libraries must be present to be able to execute the file. To transfer these imports into Q&A format to use them as evaluation datasets, the OpenAI API[4] was utilized. The following Secs. discusses the AST algorithm used for dependency comprehension dataset generation.

```
Procedure analyse(directory)
    files_list ← [];
    foreach (root, dirs, files) in os.walk(directory) do
        foreach file in files do
            if file.endswith(".py") then
                file_path ← path.join(root, file);
                file_analyzer ← FileAnalyzer(file_path, directory);
                dependencies ← file_analyzer.analyze();
                files_list.extend(dependencies);
            end
        end
    end
    return files_list;
```

Algorithm 1: DirectoryAnalyzer Class

4.2.1 AST Algorithm

Analyzing the dependencies of files in Python files is not trivial. The keyword `import` or `from` is used in Python to import an artefact. The algorithm identified four types of

[4] https://chat.openai.com

imports: complete library imports, imports from libraries, complete file imports, and imports from files. Identifying the type of the imported artefact for the categories imported from the library and file is possible. This artefact type can be a class, function, or assignment, which helps to identify the type of object imported into the file. The algorithm provides information on each Python file in the repository, including the file name, import category, and artefact name. The analysis involves a `DirectoryAnalyzer` (Alg. 1) to evaluate directories and a `FileAnalyzer` class (Alg. 2 and Alg. 3) to analyze individual files.

The DirectoryAnalyzer class is shown in Alg. 1 and is designed to systematically analyze a given directory's contents. Upon invocation of the analysis procedure with a specified directory as input, the algorithm initializes an empty list to store the results. Utilizing the `walk()` function from the `os` package in Python, the algorithm traverses through the directory hierarchy from the top-down, iteratively examining each file encountered. For files with a ".py" extension, the algorithm constructs the full file path and instantiates a FileAnalyzer object to analyze the file further. The dependencies of the file are then retrieved through the analysis method of the FileAnalyzer object, and these dependencies are appended to the list of results. Finally, the algorithm returns the accumulated list of file dependencies, providing insights into the interdependencies within the directory's Python files.

The FileAnalyzer class extracts the dependencies from the Python files, as depicted in Alg. 2 and Alg. 3. Upon invocation of the analysis procedure with a file object as input, the algorithm first reads the content of the file and initializes an empty list to store samples. Subsequently, it iterates through the Python code's AST representation, identifying import statements. Depending on whether the import is of the `form import module` or `from module import ...`, the `process_node` procedure is called to extract the relevant dependency information. This information includes the imported library name, the category of import (either "file_import" or "library_import"), and the file path of the imported module.

The `process_node procedure`, implemented within the same class, is responsible for processing individual AST nodes corresponding to import statements. It discerns the library name and import category, retrieves the file path of the imported module, and appends this information to the list of dependencies. Furthermore, the `get_artefact_type` procedure, also part of the FileAnalyzer class, determines the type of artefact defined in the Python file (e.g., function, class, variable) by traversing the AST and inspecting its structure.

```
Procedure analyse(file)
    code ← file.read();
    dependencies ← [];
    forall node in ast.walk(tree) do
        if isinstance(node, ast.Import) then
            dependencies ← process_node(node, "direct");
        end
        else if isinstance(node, ast.ImportFrom) then
            dependencies ← process_node(node, "from");
        end
    end
    return dependencies;
Procedure process_node(node, category)
    library_name ← node.name if isinstance(node, ast.Import) else node.module;
    import_category ← "file_import" if is_file_import(library_name) else "library_import";
    dependencies ← [];
    foreach alias in node.names do
        artifact_type ← get_artifact_type(alias.name, file_path);
        file_path ← get_file_path(library_name);
        dependencies.append([library_name, import_category, file_path]);
    end
    return dependencies;
Procedure get_artifact_type(node, file)
    code ← file.read();
    forall n in ast.walk(tree) do
        if isinstance(n, ast.FunctionDef) then
            return "function";
        end
        else if isinstance(n, ast.ClassDef) then
            return "class";
        end
        else if isinstance(n, ast.Assign) then
            forall target in n.targets do
                if isinstance(target, ast.Name) then
                    return "variable";
                end
            end
        end
    end
    return "unknown";
```

Algorithm 2: FileAnalyzer Class

```
Procedure is_file_import(module_name)
    if not module_name then
        return False
    end
    if module_name.startswith(".") then
        full_path ← path.join(module_path, file_path);
        return os.path.exists(full_path);
    end
    search_pattern ← path.join(root_directory, "**", module_name + ".py");
    matching_files ← glob(search_pattern, recursive = True);
    if matching_files then
        return True
    end
    return False
```

Algorithm 3: FileAnalyzer Class (continue)

Additionally, the `is_file_import()` function, presented in Alg. 3, aids in determining whether an import statement refers to a file within the project directory or an external library. This function evaluates the module name and checks if it corresponds to a file within the project directory structure. If the module name starts with a dot (indicating a relative import), it constructs the full file path and checks its existence. Otherwise, it searches for matching files within the project directory using a specified search pattern.

The analysis of the Spyder IDE repository revealed that it has 7,907 dependencies. The data shows a significant difference between the types of imports used. The project heavily relies on libraries, with 3,305 instances sourcing the whole library and only 27 instances sourcing the whole files directly. This suggests that the project prefers to use external resources instead of local file dependencies. Furthermore, the dataset indicates that 686 files were used in the project, indicating that the project operates at a moderate scale. When examining only the imports from files, the imports are mainly classes, with 1,265 occurrences, followed by functions, with 1,048 instances, and assigns, with 569 instances. Additionally, the algorithm failed to predict the correct artefact type in 140 instances where the artefact type was unknown. This distribution highlights the predominant use of classes and functions.

You are an Assistant to create question answer pairs for a programming repository. You will receive a table with information about all used imports and files of one file of a programming repository. Your task is create a short question and answer pair about the table. Vary the question so that you are ask for only one specific row sometimes about the whole table. Please either ask about imported libraries or imported files, orientate on the category column. Also write questions where the answer is No or the questions ask for a library that does not exist. If you ask multiple question in one prompt always provide the file name.
Example Question could be (FILL <<>> with data):
- Which libraries are used in the file <<FILE_NAME>>?
- What libraries are imported directly in the file <<FILE_NAME>>?
- Does the file <<FILE_NAME>> also uses the library <<LIBRARY_NAME>>?
- Is the <<MODULE>> part of the the file <<FILE_NAME>>?
- Are the files <<FILE_NAME>> and <<FILE_NAME_2>> highly coupled?
- What library does the function <<FUNCTION_NAME>> belong to in the file <<FILE_NAME>> within the programming repository?
- Is the file <<FILE_NAME>> depending on the module <<MODULE>>?

Figure 4.4: System Prompt for Creating Question-Answer Pairs

4.2.2 Data Aggregation

The raw dependencies were processed further using the OpenAI API using the "gpt-3.5-turbo-1106" model. The temperature was set to 1.5 to ensure creativity in the creation process, the maximum token limit was 256, and the top p-value was set to 1. The frequency and presence penalties were set to 0. These parameters were carefully selected to create diverse, contextually relevant questions and concise, coherent responses within specified token limits. To ensure that good Q&A pairs are built, a system prompt must lead to the desired result. Fig. 4.4 presents the system prompt for generating the Q&A pairs. Before generating the pairs, the assistant was instructed to create questions that could be answered with a "no". This ensured that guessing the most common libraries would not be a viable solution. Example questions were provided to help guide the model, such as asking which libraries were used in a particular file or where a function belongs to a particular library.

1,319 Q&A pairs were generated using the OpenAI API from 686 unique file names. Despite several attempts to modify the prompt to yield only one question and answer, the API often returned several questions and answers for a single request.

To ensure the quality of the dataset, a final set of 135 Q&A pairs was randomly chosen and manually verified for correctness. This was done by cross-checking the repository's source code to ensure that the questions and answers were correct and made sense. The random selection process was implemented to minimize the manual effort required for verification.

4.3 Meta-Information Comprehension

This thesis introduces the concept of meta-information comprehension, which refers to the ability to understand general information about a repository, such as its purpose, features, documentation, license, and contribution opportunities. An expert annotator created 50 Q&A pairs using a self-made web application, which differed from the one used for code semantics. First, all files with the suffixes *.md*, *.txt*, and *.yml* were extracted, resulting in 29 files that included meta information. Each file was reviewed and analyzed for important information relevant to the repository. The expert annotator has concentrated on relevant information, such as installing the repository, available and supported versions, and the rules for contributing. If information appeared relevant, a Q&A pair was created in the provided text areas for questions and answers and added to a text file. The file name and the module in which the file was located were also saved.

4.4 Data Analysis

To verify the difference between the different dataset modalities, which is a prerequisite to evaluating the performance of the trained models, a detailed data analysis and a comparison was undertaken.

4.4.1 Typical dataset examples

For a first impression of the differences between the three dataset modalities, typical samples are shown in Fig. 4.5, Fig. 4.6 and Fig. 4.7. Based on the provided samples, it is evident that the Q&A pairs for the source code semantic dimension are primarily concerned with the meaning of the source code. To answer these questions, the model must thoroughly understand the source code's syntax and semantics to allocate the necessary information accurately. Moreover, the examples demonstrate that the dataset seems heterogeneous since it focuses on various source code parts.

Question: In file script.py in module app.tests, what is arr and what is it set to?
Answer: arr is a numpy array initialized with the list li, which is set to [1,2,3] before.

Question: Why is there a todo in this snippet and what has to be done so that it can be removed?
Answer: According to the TODO comment there is a section needed until all core plugins are migrated. Then it can be removed.

Question: What is the task of test_generate_cookiecutter_project_defaults?
Answer: The task is to test the function generate_cookiecutter_project.The test case creates some temporary files. and then calls generate_cookiecutter_project with the path to these temporary files. Finally, it is tested that value is in the result and the status is true. This probably indicates that the cookiecutter project was successfully initiated.

Figure 4.5: Three example Q&A pairs for the source code semantic modality

In comparison, the Q&A pairs of the dependency dimension are different. The questions are always focused on libraries, modules, or files. Generally, the answers for this dimension are shorter, less about providing a description and more about producing verifiable information.

In the last dimension, the Q&A pairs from the meta-information dimension in Fig. 4.7 focus more on general information about the repository. The questions can be short, such as in the first example, or longer when asking an open-ended question, as seen in the second example. This dataset is the most challenging for the models because the format and type of the Q&A pairs are pretty different.

Question: Which libraries are directly imported in the file "spyder/plugins/ipythonconsole/utils/manager.py"?
Answer: The directly imported libraries in the file "spyder/plugins/ipythonconsole/utils/manager.py" are: os, signal, psutil, and asyncio.

Question: Is the module qtpyQtCore instantiated in the file edgeline.py?
Answer: No, there is no instantiation of the module qtpyQtCore in the file edgeline.py.

Question: What file is imported from the file 'decoration.py' within the programming repository?
Answer: spyder/utils/palette.py

Figure 4.6: Three example Q&A pairs for the Dependencies modality

Question: What version of Python is required to run Spyder?
Answer: Spyder requires Python 3.8 or higher as its core language.

Question: What are the main components of the Spyder IDE?
Answer: The software has five main components:
1. Editor: Multi-language editor with real-time code analysis tools, automatic code completion, and go-to-definition.
2. Interactive Console: IPython consoles with workspace, debugging support, and inline plot rendering.
3. Documentation Viewer: Real-time documentation rendering with Sphinx for classes and functions.
4. Variable Explorer: Inspect any variables, functions, or objects created during your session.
5. Development Tools: Static analyzer, interactive debugger, profiler, project support, file explorer, and full regex search.

Question: What is the first step to be taken after releasing a new version of Spyder?
Answer: The first step is to publish the release on the GitHub Releases page. This involves copying the contents of the previous release description, updating relevant information and links to point to the new Spyder version and changelog entry, and editing the previous release description to only have the changelog line.

Figure 4.7: Three example Q&A pairs for the Meta Information modality

4.4.2 Token & Sentence Analysis

To better understand the dataset, this subsection describes the further analysis of the Q&A pairs on token and sentence levels. Fig. 4.8 shows the number of tokens used for each question per dataset dimension. On the y-axis, the number of tokens is displayed, while the x-axis displays the different dataset modalities. The plots show that the median (solid line) and the mean (dotted line) are slightly the same in all three modalities. A one-way analysis of variance (ANOVA) was conducted to test whether the distributions of the datasets were different. The ANOVA revealed no significant effect of the factor 'Dimension' on the dependent variable number of tokens ($F(2, 322) = 1.91$, $p = 0.15$). This leads to the conclusion that the length of the questions is consistent across all datasets.

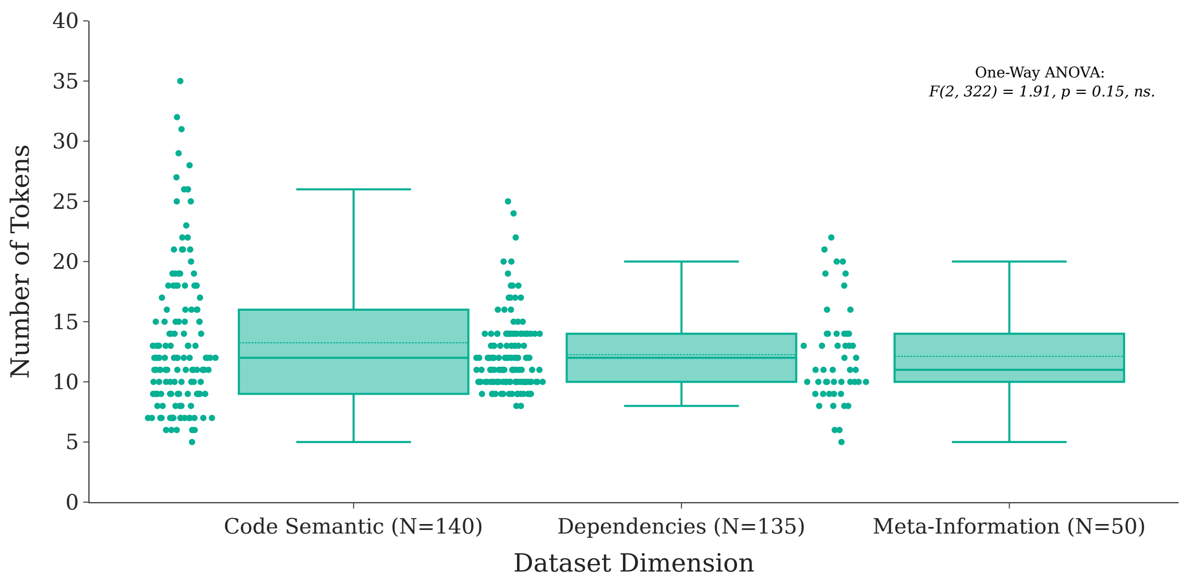

Figure 4.8: number of tokens per question and split per dataset.

Compared to that, the number of tokens in the answer is differently distributed. Fig. 4.9 shows the number of tokens per dimension. Descriptively, the distributions of the datasets show differences. While the source code semantic and the dependencies dimension have many short answers and a few longer answers, the variance in the meta-information dimension is much larger. As the plot suggests, the ANOVA revealed a significant effect of the 'Dimension' factor, indicating that the number of tokens in the datasets differs significantly ($F(2, 322) = 58.55$, $p < 0.001$, $\eta^2 = 0.266$).

The η^2 effect size indicates that the effect can be classified as a small effect following the convention of Cohen [109]. In practical terms, the model needs to generate answers of different lengths based on the varying lengths of the modality's answers, which can be more challenging than constantly generating the same length.

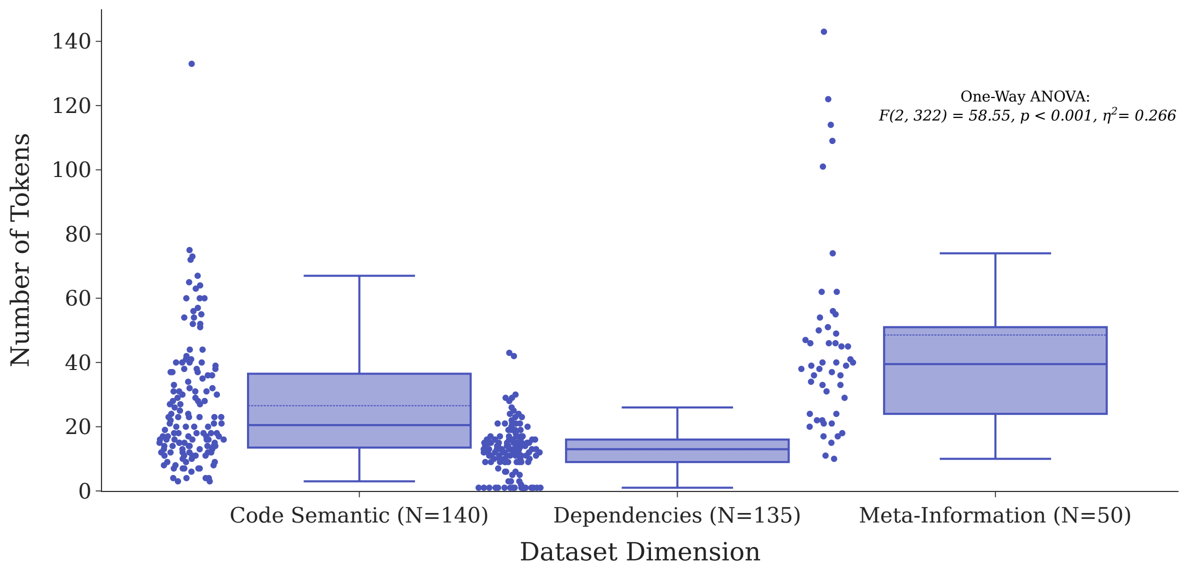

Figure 4.9: number of tokens per answer and split per dataset.

Another analysis was done for the number of sentences for each dimension. Fig. 4.10 shows the mean number of sentences per question and answer, with error bars representing the standard deviation of the distribution. Compared to the number of tokens with a slight but existing variance, all distributions of the sentences showed a small to non-existent variance. There were questions with more than one sentence in the code semantic dimension, while only single-sentence questions existed in the dependencies and meta-information dimension. Another one-way ANOVA was conducted on the dimension factor for the number of sentences in the questions, revealing a significant effect on the 'Dimension' factor. This indicates that the number of sentences is significantly different between the three modalities ($F(2, 322) = 9.99$, $p < 0.001$, $\eta^2 = 0.058$). However, the low variance of the distributions of dependencies and meta-information must be considered in the interpretation, and the very low effect size should be approached with caution.

The analysis of the number of sentences in the answers reveals some differences. The dimension dependencies necessitate that answers be single-sentence only, in keeping with the questions. However, the variance of the code semantic and meta-information answers shows a significant difference from the questions. This is in line with the results from the number of tokens, which show that the answers from the meta-information are very different in length and contain very different numbers of sentences. The conducted one-way ANOVA revealed, as well as for the answers, a significant effect on the factor 'Dimension' (($F(2, 322) = 21.05$, $p < 0.001$, $\eta^2 = 0.116$). As with the questions, the low variance of the modalities is a problem for the interpretability of the ANOVA, as individual samples have a significant influence. Nevertheless, according to Cohen's convention, with an effect size of $\eta^2 = 0.116$, the effect can be classified as small following the convention of Cohen [109].

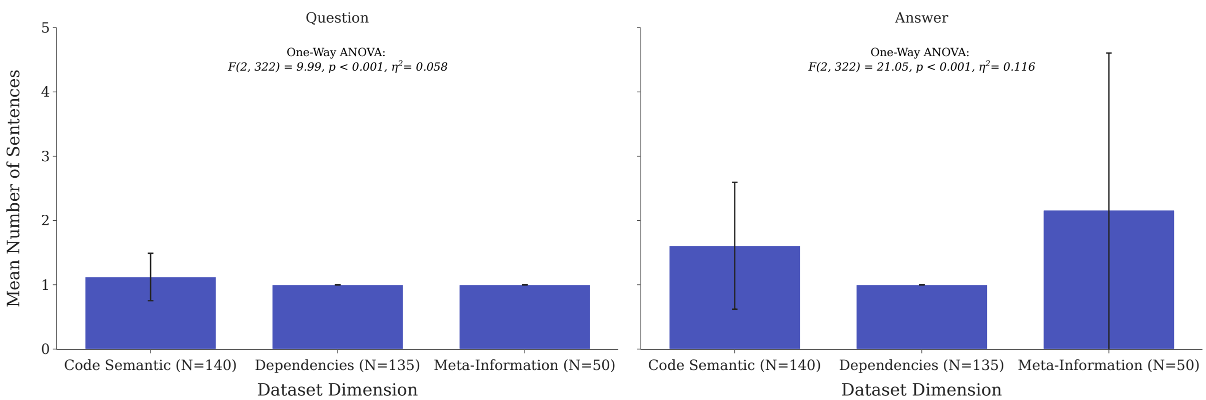

Figure 4.10: distribution of question keywords per dataset dimension

4.4.3 Question Keyword Diversity

Another noteworthy observation from the dataset pertains to the percentage of question keywords used in each dimension, as illustrated in Fig. 4.11. For each Q&A pair question, the question keywords were extracted using a regex that extracts commonly used question keywords. The extracted keywords are What, Which, How, Why, Where, Who, From, and When. According to the source code, around 50% of the questions use 'What' as a keyword, and nearly 20% use 'Which'. This suggests that, in most cases, the questions ask for a subject or object. It is also interesting to note that 12% of the questions use multiple keywords. In such cases, the dataset sample aims to achieve two different objectives. For example, "Which parts need to be instantiated in the constructor of the class ApplicationsDialog, and what is the task of parts?".

Regarding the dependencies dimension, question keyword distribution differs from the previous dimension. In this case, only 26.7% of samples use the word "What," 25% use "Which," and 32.6% do not use any original question keyword. Most of the samples in the "Others" category utilize indirect questions such as "Is the..." or "Does the...". The distribution of meta-information differs significantly from the distribution of the other two modalities. In the former, there is a high concentration of the question word 'What', which accounts for 68% of the overall questions. On the other hand, 10% of the queries still employ 'How' as a question word, typically when asking for instructions or a 'how-to' guide. Overall, the distribution in all three modalities is noticeably distinct, and this explains the challenges the model faces in performing well across all modalities.

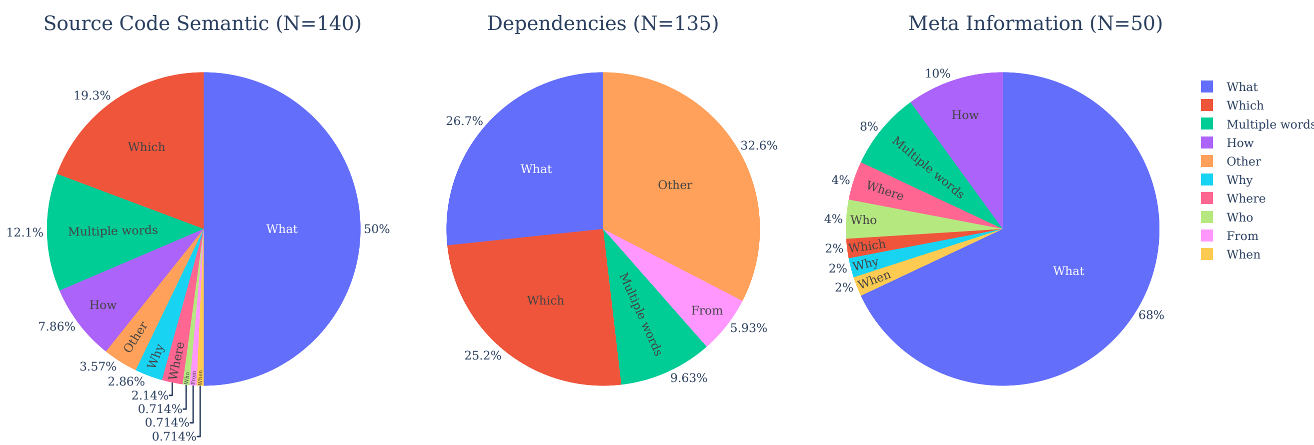

Figure 4.11: Mean number of sentences per dataset dimension.

4.5 Challenges & Limitation

This section outlines the challenges and limitations regarding the created dataset. Despite the best conscience and many considerations, the following things must be considered if the dataset should be used as an evaluation dataset.

Small dataset size The SpyderCodeQA dataset consists of 325 samples, which is a significant number. In comparison, benchmarks like HumanEval only contain 164 samples. Considering the time and resources required for manual sample creation is essential. Other datasets in this research area, such as CS1QA [90] with 9k samples and CodeQA [88] with approximately 200k samples for the code comprehension task, have much larger sample sizes than the dataset introduced in this thesis. However, these datasets were not manually verified and only created from online data sources, ensuring a lower data quality. Nevertheless, the generalizability and value of the dataset could be higher.

Different Knowledge Level of Creators For the source code semantic dimension, the Q&A pairs were created by humans. While the number of participants was with ten people relatively low, also the knowledge level of the participants about the Python programming language and the working experience was high. That could lead to a bias in the difficulty of the questions asked. Assuming you want to test whether a model can answer simple questions for beginner programmers, the questions from the semantic dimension may not necessarily be helpful and accurate.

Unknown repository The individuals who took part in the study considered themselves experts in Python. However, none of them had previously contributed to the Spyder IDE repository, which means that none of the participants were experts in this specific code base. Although this may not pose a disadvantage, it does suggest that the questions and answers provided may not be as in-depth as those provided by a Spyder IDE contributor.

Low heterogeneity of the Q&A pairs in dependency dimension The Q&A pairs in the source code semantic dimension have a great variety, but the ones generated automatically in the dependency dimension are often very similar. This is to assess the model's ability to answer these questions accurately. However, a more comprehensive range of questions would be preferable to test the model's performance as a coding assistant. Therefore, a further improvement of the dataset would be adjusting the model's system prompt that generates the Q&A pairs or developing a new way to measure the dependencies of the different repository components.

Only 1-hop Dependencies The relationship between the two source code files is adequately described using the dependencies dimension. However, the dataset dimension lacks a mapping that goes beyond linking two files. Therefore, it would be beneficial to devise a way to create 2-hop or even n-hop structures that the models can comprehend.

Meta-information dimension quality is not measured The quality of the source code semantic dimension dataset was ensured through a rating process conducted by participants. The dependency pairs were also manually verified to be correct. However, the meta-information dimension lacks quality testing. The Q&A pairs were created exclusively by the author of the thesis, which could introduce bias in the formulation of the questions and answers and the selection of information to create the pairs. This dimension may not be as objective as others, as different people may create different pairs.

5. Method

The methodology section provides a comprehensive overview of the methods and techniques used to address the research questions. To begin with, in Sec. 5.1, an explanation of the data preprocessing procedure that involves generating chunks from the repository is presented. This transformation in the correct format is essential so that the Self-Alignment can generate training datasets and the vector database of the RAG pipeline can be used effectively. An overview of the model structure of Mistral 7B is presented in Sec. 5.2. This is followed by a detailed explanation of the Self-Alignment pipeline that is the fundament of the fine-tuning approach in Sec. 5.3. How the chunks were stored in the vector database and how they were queried in the RAG pipeline is described in Sec. 5.4. The chapter concludes by explaining the LLM-as-a-judge approach in Sec. 5.5, which is the evaluation method used to measure the performance of the models in this thesis.

5.1 Data Preprocessing

The source code needs to be preprocessed to create a coding assistant for the Spyder IDE repository. This involves formatting all files to fit the desired structure for training models using Self-Alignment or creating a vector database for the RAG pipeline. A pipeline was created to preprocess source code files and other data into suitable samples, as shown in Fig. 5.1. The repository was fetched at commit `0f8398a`, a version of the release of Spyder `5.x`. First, all files from the repository were loaded using individual loader classes for each file type. Ensuring every file was represented correctly and preserved in the correct format was essential. After loading the documents, they were divided into smaller chunks using a splitter that uses individual separators per file type. For example, for Python files, the separators were: `\nclass,` `\ndef`, and `\n\tdef`. Whereas for Markdown files, they were: `\n#{1,6}`, `` ```\n ``, `\n\\*\\*\\*+\n`, `\n---+\n`, `\n___+\n`, `\n\n`, `\n`. The text was divided into chunks of a maximum of 1500 characters, with each chunk overlapping by 200 characters, based on the blog post by Rubens Zimbres[1].

The code chunks are further processed using the Metadata Extractor as shown in Fig. 5.1. This component extracts all the available metadata for each chunk of code. The metadata includes the file name, module, and flag indicating if the code chunk contains a class or function. The start and end lines of the code chunk are also

[1] https://shorturl.at/u0bdA

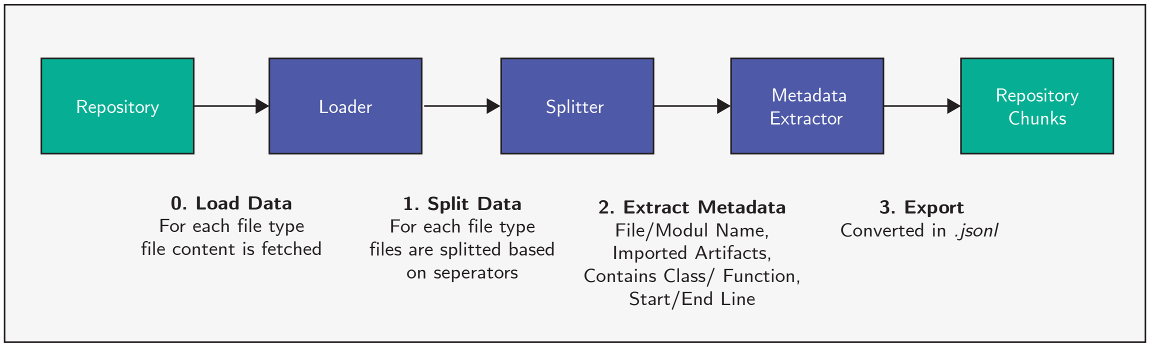

Figure 5.1: Data Preprocessing Pipeline to generate data samples. First, all files from the Spyder IDE repository are loaded with individual Loader classes. That is followed by a Splitter, which splits the documents into chunks depending on the file type. All chunks are then further processed into the Meta Data Extractor, which creates all metadata. The metadata is added to the chunks in the last step and exported in a *.jsonl* file.

included. All the imports from the file were also added to the metadata to ensure that each code chunk had information about the imported artefacts. This ensures that all the external artefacts are visible, regardless of whether the code chunk is in the middle or at the end of a file. In the final step, the extracted metadata were added to the chunks and saved as *.jsonl* file and uploaded into Huggingface.[2] Fig. B.1 shows an example chunk for a Python file and Fig. B.2 for a Markdown file in Appendix B.

5.2 Model Details

This section gives an overview of the architecture of Mistral 7B [22]. It is used as the base model for fine-tuning and interacting with the RAG pipeline. It was chosen because it outperforms Llama 2 7B, 13B [1] and CodeLlama 7B [1] on almost every benchmark and is, therefore, one of the best open-source models that can generate natural language and code at the same time. Mistral 7B is based on the transformer architecture [38] and, using similar SOTA techniques like pre-normalization using RMSNorm, SwiGLU activation function and rotary positional embeddings that were used in the architecture of Llama [11] and Llama 2 [1].

In addition to that, Mistral 7B utilizes two attention mechanisms: Grouped-

[2] https://huggingface.co/datasets/pesc101/spyder-ide-respository-raw-chunks

Table 5.1: Model Details extracted from the original implementation [22].

	Dimension	Layers	Head Dimension	Hidden Dimension	N Heads
Mistral 7B	4096	32	128	14336	32

Query Attention (GQA) [110] and Sliding Window Attention (SWA) [111]. These mechanisms enhance performance and efficiency in various ways. GQA accelerates inference and reduces memory requirements during decoding, enabling larger batch sizes and greater throughput. SWA is a more efficient and cost-effective way of handling longer sequences, often a limitation in LLMs. Combining these attention mechanisms significantly improved the model's performance and efficiency compared to the Llama family models. Furthermore, Tab. 5.1 shows the model architecture parameters.

5.3 Self-Alignment

This section presents the conception and implementation of the Self-Alignment pipeline, which is necessary to answer the research question **RQ1** defined in Sec. 1.1. Because it can be difficult and time-consuming to create Q&A data for source code files, and suitable datasets are often unavailable or non-existent, using data augmentation to create a dataset is a worthwhile solution. The Spyder IDE's files are available in chunks after data preprocessing (as described in Sec. 5.1). Still, for the model to learn to answer questions about them, it is necessary to use samples in the form of Q&A pairs as training data. Therefore, Self-Alignment is used in this thesis to generate the required training data as described in Sec. 3.2. This thesis approach differs from Self-Alignment cause it consists of several sequential prompt templates to help the model generate the most accurate Q&A samples possible, as described in Fig. 5.2.

Self-Augmentation The repository's code chunks are passed as input into Mistral 7B in the first step. In addition, a question is selected from a question corpus from Chap. 4 and is also used as the input to the model. The questions are categorized into three categories to reflect the dimensions of a repository, as explained in Chap. 4. The list of used questions can be found in Appendix C in Fig. C.1 for Code Semantics, in Fig. C.2 for Dependencies and in Fig. C.3 for Meta-Information. The system prompt that generates the teacher data D_0 is in Appendix D on Fig.

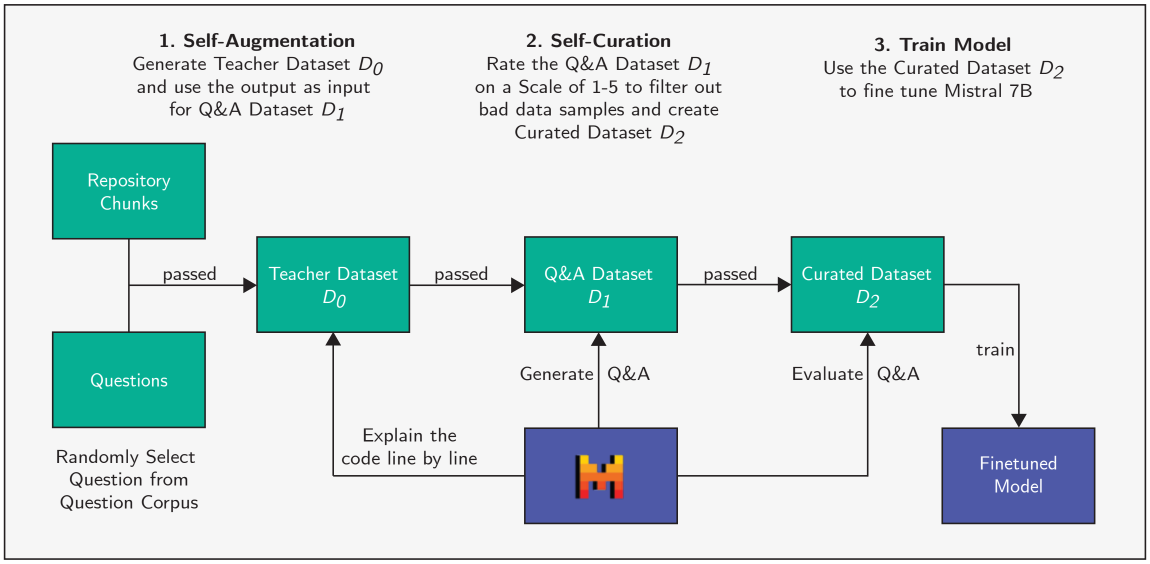

Figure 5.2: Overview of the Self-Alignment pipeline inspired by Li et al. [29]. **1. Self-Augmentation:** The repository chunks and a randomly selected question from the question corpus are combined in the System prompt (Shown in Appendix D in Fig. D.1). Mistral 7B generates the Teacher Data D_0 based on this. The output is further passed, and Mistral generates the Q&A Dataset D_1. **2. Self-Curation:** Q&A pairs are curated on a scale of 1-5 to ensure data quality and filtered to create the final Curated Dataset D_2. **3. Train Model:** The Curated Dataset D_2 is used to fine-tune Mistral 7B.

D.1. The prompt is designed to explain each line of code individually so that the model can generate as much context and semantic information about the source code as possible. The output is a natural language description mixed with source code explaining the repository chunk.

In the second step, the output generated in the previous step is passed back into Mistral as input, generating the dataset D_1. The dataset needs to be converted into Q&A format to create an aligned Instruction model despite the generated data being full of code explanations. Mistral's task is to generate Q&A pairs from this source code explanations. The system prompt for the Q&A data is shown in Fig. D.2. It instructs the module to include file and module names to ensure the model always knows the file the question aims for. The prompt also specifies that code should be added to the answer. The assumption is that answers regarding a repository should include code, not only natural language, to improve the model's ability to generate code. First, the model's output format is specified to include "Question:"

and "Answer:" tags for generated questions and answers, respectively. This helps to parse the strings afterwards and extract only the Q&A pairs from the output.

Self-Curation To generate high-quality training data, the last step of the pipeline involves curating the data samples, referred to in Fig. 5.2 as D_2. As demonstrated by Li et al. [29], generating data samples without criteria can result in poor or inappropriate training data. To address this issue, they proposed Self-Curation, in which the base model Mistral 7B rates the data samples from D_1. Therefore, the model evaluates the Q&A pairs on a scale from 1 to 5 in this step. The system prompt is displayed in Fig. D.3. The model evaluates whether the response is an excellent example of how an AI Assistant should respond to user instructions. A score of 1 indicates that the answer is incomplete, not precisely what the user asked for, or off-topic. In contrast, a score of 5 represents a perfect answer from an AI assistant that is structured and thoroughly answers the user's question. All examples with a score lower than four are removed from the dataset.

Summary The Self-Alignment pipeline was executed for the 7943 chunks generated as described in Sec. 5.1. That included all source code, config and markdown files. It is important to note that the pipeline to generate Q&A examples can be executed multiple times in a row. A detailed analysis of generating for each chunk multiple Q&A is further discussed in Sec. 6.5.1.

5.4 Retrieval-Augmented Generation Pipeline

This section presents the conception and implementation of the RAG pipeline to answer the research question **RQ2** as defined in Sec. 1.1. The main idea is that questions about the repository can be answered with the proper context, e.g., the right chunk of information. Therefore, the RAG pipeline consists of a vector database storing the code chunks in a vector representation. The vector representation allows for comparison and for determining semantic similarities between chunks. If two chunks have the exact words or semantics, the similarity score is high; the similarity score is low if the two chunks are less similar. This enables the search for similar chunks and their retrieval, which the model can use to generate the desired output. The implemented RAG pipeline in this thesis is illustrated in Fig. 5.3. Like in the Self-Alignment pipeline, the preprocessed chunks are used. In this case, they were transformed to a 768-dimensional vector representation using the embedding model Instructor [35]. Compared to other models, is Instructor [35] a multitask model

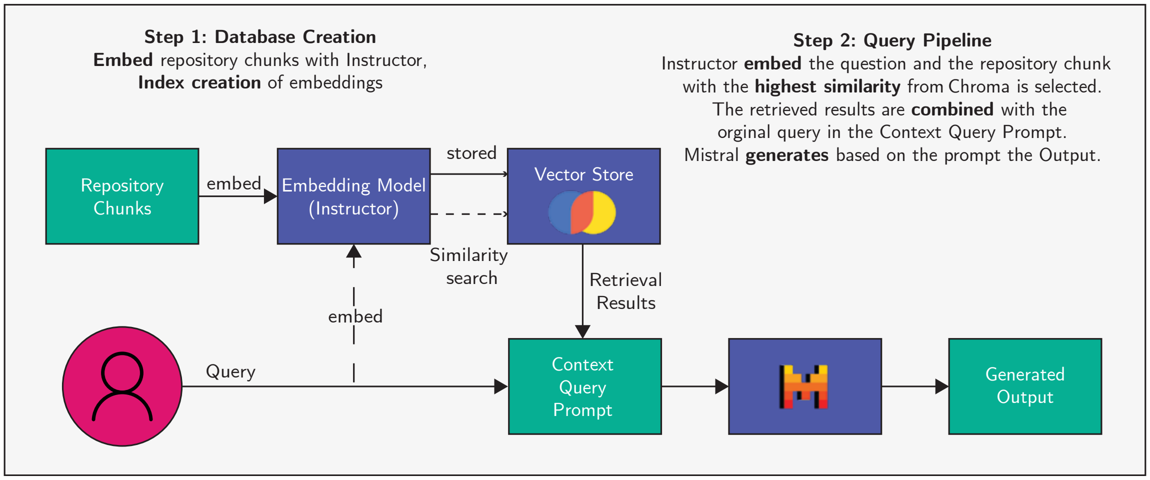

Figure 5.3: Overview of the RAG pipeline inspired by Lewis et al. [13]. **1. Database Creation:** Source Code files are embedded using Instructor [35]. Chroma is initialized and indexed with the embedded source code files. **2. Query Pipeline:** queries are transformed into embeddings following the dotted line. Then, through a similarity search, the n-chunks are retrieved and combined as Context Query Prompt. That prompt is passed to the generator, which produces the user response.

that can be used without fine-tuning on several downstream tasks, e.g. retrieving code snippets. It is a pre-trained model with 110 million parameters that generate embeddings for retrieval, classification, or semantic search tasks.

Initialization The data is stored in the in-memory version of Chroma[3], an optimized database for storing vector representations, e.g. embeddings and provide an interface to query the chunks efficiently. The database is initialized by assigning an ID to each chunk and indexing the metadata, such as file name, module name, start and end line of the chunk and the associated imports. The creation of the index, which included the calculation of the embeddings, took approx. 25-30min. This ensures a quick response time and enables data retrieval based on metadata queries.

Retrieval Once the vector database is initialized, the user can send requests. Whenever a user sends a request to the RAG pipeline, the request goes through several steps shown in Fig. 5.3. First, the request is transformed into a standardized 768-dimensional vector representation using Instructor. Then, each chunk's simi-

[3]https://www.trychroma.com

larity score (cosine similarity) is calculated to retrieve the chunks with the highest similarity from the vector database. Furthermore, filters such as the file and module names are applied to minimize the search scope. The n-chunks with the highest similarity are then retrieved, and the original content of the chunk is passed on to the generator.

Generation In the final step of the process, the base model Mistral 7B acts as the generator that produces the response on behalf of the user. The system prompt generated the response is displayed in Fig. D.4. Initially, the model is instructed to generate an answer based on the task context. Then, based on the specific number of requested chunks n, all the chunks are appended to the prompt. Finally, the user's question is added to the prompt. The generator then creates the response from the prompt template, which is forwarded to the user.

Summary This pipeline stores the preprocessed chunks into a vector representation and can retrieve them based on the user's query. The LLM generates the user's response based on the retrieved context. The expectation is that this helps the LLM better provide suitable answers to the user's requests.

5.5 LLM-as-a-judge

This section presents the conception and implementation of the LLM-as-a-judge approach to evaluating model performance, e.g., to answer the research questions. Evaluating coding assistants is challenging because of their vast capabilities, the inadequacy of existing benchmarks in measuring human preferences, and the time required to complete this evaluation manually. The judgement is made even more difficult because the questions can be answered differently but can still be correct. This applies particularly to writing code and answering questions about a repository, as this thesis will examine. Therefore, the performance of models on the Q&A evaluation dataset created in Sec. 4 was evaluated in pairwise comparison using strong LLMs (primarily GPT 3.5/4) as judges [37].

Fig. 5.4 shows the thesis's model-based pairwise comparison pipeline. It is inspired by the proposed approach of Zheng et al. [37]. For each Q&A pair in the evaluation dataset, the two models M_1 and M_2 receive the question of the Q&A pair. The models follow the prompt template shown in Fig. D.5. Firstly, the model is directed to behave like a coding assistant and to provide answers to questions related to the Spyder IDE repository. The models are then provided with general

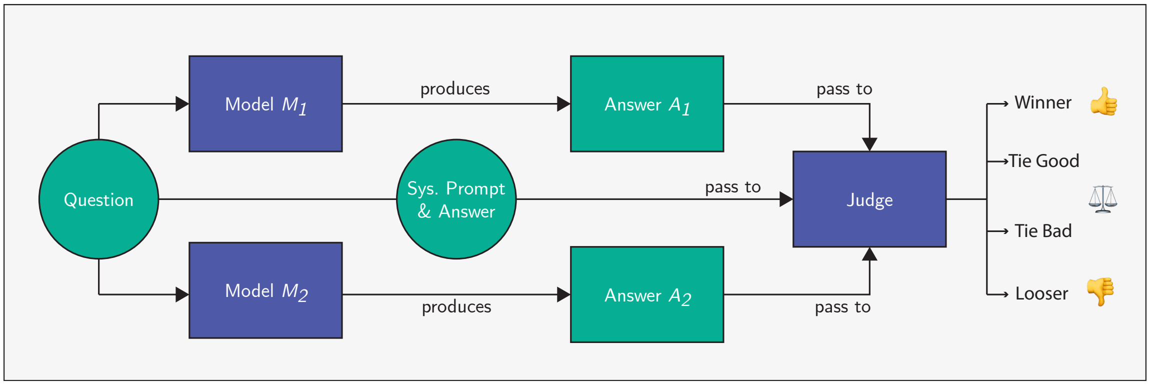

Figure 5.4: Overview of the LLM-as-a-judge pairwise comparison evaluation [37]. The LLMs M_1 and M_2 are tested against each other using questions from the evaluation dataset. Both models answer the related question, described here as A_1 and A_2. The judge (GPT-3.5) receives the system prompt shown in Fig. D.6 that includes the original Question, the correct Answer, both answers of M_1 and M_2, and the instruction to judge the quality of both answers and determine the outcome as winner, tie good, tie bad, or looser.

information about the repository and reminded always to answer honestly and not provide false information. Finally, the evaluation dataset question is given as input to the models, and the models generate the answers A_1 and A_2.

After the models generate the answers, the judge is prompted to judge them. The prompt template for the judge is shown in Fig. D.6. The judge model is instructed in the system prompt to act as a judge to evaluate the quality of responses provided by two AI assistants. It is given instructions on how to evaluate and is required to give the output in the format: [[A]], [[B]], [[C]] or [[D]] to indicate its decision. The output [[A]] signifies that Model A generated the better response, while [[B]] indicates that Model B generated the better response. If the judge decides on [[C]], it means that both answers are equally good (Tie Good), and if the judge decides on [[D]], it means that both answers are incorrect (Tie Bad). The judge then receives the question with the related correct answer, and the generated answers are A_1 and A_2. As input, the question (labelled as "User Question" in the prompt), the answers (labelled as a "Model Solution" in the prompt), and both answers A_1 and A_2 are inserted. To ensure clarity, each piece of information is enclosed within square brackets, with an identifier indicating the type of information it contains.

Position Bias In their original paper, Zheng et al. [37] demonstrated that position bias can be a significant limitation when using LLM-as-a-judge. This bias refers to the LLM's tendency to favour certain positions over others, which can result in potentially biased decisions. It is crucial to address this bias to ensure that decision-making is fair and objective. It is worth noting that position bias is not unique to machine learning and can also be observed in human decision-making [112]. Therefore, as suggested by the paper, the order of models was randomized for each Q&A pair to prevent any bias towards one of two models used in a dataset. This was done to ensure that a model was not rated better simply because its answers were presented at the beginning or end.

6. Experiments

This chapter describes the experiments conducted to measure the performance of the developed methods and compare them with existing approaches. The objective is to evaluate the LLM Self-Alignment method, the RAG pipeline, and their combination to determine how much the RAG pipeline contributes to the final score. The first Sec. 6.1 gives an overview of the benchmarks HumanEval and MBPP that are used to measure the code generation abilities (Subsec. 6.1.2). This is followed by the implementation details of the LLM-as-a-Judge evaluation (Subsec. 6.1.1), which measures the performance on repository code Q&A.

To address the first research question (**RQ1**), comprehensive information on the experiments focusing on Self-Alignment can be found in Sec. 6.2. This includes a detailed overview of the training parameters in Subsec. 6.2.1, the performance on the SpyderCodeQA dataset (Subsec. 6.2.2), and the benchmarks HumanEval and MBPP (Subsec. 6.2.3). In response to the second research question (**RQ2**), a presentation of all the details for the RAG pipeline is available in Sec. 6.3. This section will overview the experiment structure and present the results on the SpyderCodeQA evaluation dataset (Subsec. 6.3.1). Subsequently, the combined results of both pipelines to answer the third research question (**RQ3**) are presented in Sec. 6.4.

Various additional experiments exploring different hyperparameter combinations are discussed in Sec. 6.5. The aim is to determine the optimal size of the training dataset. One experiment, detailed in Subsec. 6.5.1, focuses on generating datasets of varying sizes using Self-Alignment and comparing their effectiveness on SpyderCodeQA. Another set of experiments is described in subsection 6.5.2 to determine whether the use of a heterogeneous data set improves the performance of the model. Therefore, the effect of adjusting the temperature and Top-P parameters in the Self-Alignment approach when generating D_0 and D_1 is investigated. Finally, in Subsec. 6.5.3, an experiment concentrates on comparing the performance of the judges in the LLM-as-a-Judge evaluation. In addition to GPT-3.5, GPT-4 Turbo is used as an alternative model for judging. Finally, the chapter ends with a summary and discussion of the achieved goals in Sec. 6.7.

6.1 Metrics

This section discusses the performance measured using LLM-as-a-Judge in Subsec. 6.1.1, as well as the metrics of HumanEval and MBPP in Subsec. 6.1.2.

6.1.1 LLM-as-a-Judge

For measuring the performance of the LLMs regarding repository-level programming, the created evaluation dataset is used to execute an LLM-as-a-Judge evaluation. Each LLM-as-a-Judge evaluation was performed in pairs of two models. The base model was usually tested against a modified model (fine-tuned, RAG, or both combined) as described in Sec. 5.5. The parameters for creating the answer were a temperature of 0.7, a Top-P of 0.9, and a max token of 2500. These same parameters were used for the judge as well. The metric used is the average win rate, which is the proportion of Q&A pairs in which the judge has decided that the model's answer is better than the other or not a Tie. The average is calculated over k runs executed with the same parameters to take into account deviations of the judge, as this evaluation method is not deterministic due to the randomized distribution of the order of the models' answers and the use of an LLM as a judge. In addition to the decision for one of the two models, the judge could also evaluate Tie Good (both gave the same good answer) and Tie Bad (both answered the question incorrectly). As a last option, the evaluation could result in 'No value' when the judge does not return a parseable result in the output.

6.1.2 Benchmarks

In addition to evaluating whether a coding assistant has become better at answering questions about a repository, benchmarks aim to determine whether the models' code generation abilities have changed. Therefore, two benchmarks are used to evaluate the general coding abilities of the models after fine-tuning.

HumanEval HumanEval is a dataset introduced by OpenAI [5]. It comprises 164 handwritten Python problems, including a function signature, docstring, body, and several unit tests. This dataset is designed to evaluate the functional correctness of LLM on coding tasks. The models are given a function signature as a prompt to generate code in a zero-shot setting. $p = 0.95$ for Top-P and $t = 0.2$ for temperature were used for evaluation. The functional correctness of the LLMs is evaluated by computing their pass@k metrics with $k = 1$ and $k = 10$, using the unbiased estimator proposed by Chen et al. [5]. The pass@k considers the test successful if one of the k

samples produced passes all the tests. It is computed as follows:

$$\text{pass}\,@k := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{1}$$

where n is the total number of samples and c the number of correct samples and $\mathbb{E}$ the expected value.

MBPP The Mostly Basic Programming Problems (MBPP) dataset consists of 974 programming tasks designed to be easily solvable by new programmers [36]. Out of these, only 500 samples are used for evaluation, as the remaining samples are utilized for training or testing purposes. Each task includes a detailed description in English, a code solution, and three automated test cases. The prompt and generation process uses a few-shot setting in InCoder-style prompts. The prompt is provided to the model as a document, and only one solution is included to help the model identify the required function name for the unit tests. The prompt is structured as follows: `'\n{description}\n{test_example}\n'`. A single generation per problem (pass@1) is used, where the model only gets one chance to solve each problem. However, the approach of Chen et al. [5] is still followed, similar to HumanEval, for pass@k estimation. Solutions are generated for each problem (in this case) to estimate the success rate for a sample ($n = 10$).

6.2 Self-Alignment

This section presents the experiments done for Self-Alignment. First, the model training parameters are shown in Subsection 6.2.1, followed by the presentation of the results on the evaluation dataset in Subsection 6.2.2 and the results on benchmarks in Subsection 6.2.3.

6.2.1 Model Trainings Pipeline

This section overviews the fine-tuning training processes. As the base model, the Mistral 7B Instruct model v0.2[1] was trained on the Self-Aligned dataset. It is important to note that the dataset samples are distinct for each run of the Self-Aligment

[1] https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2

pipeline. Therefore, the following explanation of the dataset refers to one execution but can vary when executing it again.

The best results on the benchmarks and the LLM-as-a-Judge evaluation were produced by executing the Self-Alignment pipeline twice. When the whole pipeline is executed, the Self-Augmentation that creates the teacher dataset D_0 is executed twice for each of the 7943 chunks. That results in 15886 data samples that are further processed to the creation step of the Q&A dataset. The data samples are then given to the second step of the Self-Augmentation to create the Q&A dataset D_1. After the Self-Curation, 1452 samples were removed, resulting in 14434 Q&A pairs used as training data for the base model.

The base model Mistral 7B was trained using supervised fine-tuning (SFT) [30] and 4-bit Quantization Low-Rank Adapters (QLoRA) [31]. The supervised training procedure is based on the approach proposed by Ouyang et al. [30] since the dataset consists of pairs of prompts (such as questions) and outputs (such as answers). The use of QLoRA is based on the original implementation by Dettmers et al. [31] in 2023, which was utilized in the HuggingFace implementation[2]. More fundamental details on the training techniques can be found in Sec. 2.3 and 2.3.2.

The training parameters are shown in Tab. 6.1. The model has trained five epochs with batch size 32 on an NVIDIA RTX A6000 with 49GB VRAM. The computing cluster consisted of 128 CPUs and 1TB of RAM. The model was trained using BF16 precision, which reduces the model's memory consumption and improves performance and gradient checkpointing to reduce memory accumulation.

The following settings were used for the optimization: the cross-entropy loss was used as a loss function, while the Adam optimizer was used with $\beta_1 = 0.9, \beta_2 = 0.999$, following the implementation by Zheng [95]. The learning rate was set using a cosine decay scheduler, starting with an initial learning rate of $1e-4$ and a warm-up ratio of 0.03. The loss and learning rate for each step of the training runs can be found in Appendix E in Fig. E.1 and Fig. E.2. During each training run, the loss consistently decreased, with a significant drop at the end of each epoch. The learning rate also behaved as expected, with the warm-up ratio leading to an initial increase in the learning rate, followed by a gradual decrease over the training duration.

The following parameters were chosen for quantization: LoRA R and Alpha 64, following the approach of equalizing the number of R and Alpha to reduce noise, as suggested in this blog post[3]. LoRA dropout was set to 0.1, and the weights were calculated in 4-bit using normalized float-4 (NF4) for the calculation, as recommended by Dettmers et al. [31].

[2]https://huggingface.co/docs/peft/index
[3]https://shorturl.at/GvgIY

Table 6.1: Model Trainings Configuration for Fine-tuning Mistral 7B

Category	Parameter	Value
Environment	Number of GPUs	1
	Mem per GPU	49140
	Number of CPUs	128
	RAM	1TB
Model	Model parameters	7B
	Epochs	5
	Batch Size	32
	BF16 Precision	Enabled
	Gradient Accumulation Steps	1
	Gradient Checkpointing	Enabled
	Max Gradient Norm	0.3
Optimization	Loss function $\mathcal{L}$	Cross entropy
	Optimizer	Adam
	Optimizer parameters	$\beta_1 = 0.9, \beta_2 = 0.999$
	Learning rate scheduler	cosine decay
	Initial learning rate	$1e - 4$
	Warmup Ratio	0.03
Quantization	LoRA r	64
	LoRA α	64
	LoRA Dropout	0.1
	4-bit Precision	Enabled
	4-bit Compute Dtype	Float16
	4-bit Quantization Type	NF4

Flash Attention 2 [113] was used to speed up model training by a factor of 3 [113]. For the dataset with 14434 samples, the five-epoch training took four and a half hours. After the training, the LoRA layers were merged into the base model Mistral 7B to reduce the response time when using the model for inference.

6.2.2 Results on SpyderCodeQA

The following section presents the evaluation results of the fine-tuned model on the SpyderCodeQA dataset using LLM-as-a-Judge evaluation. The evaluation process involved passing 325 Q&A pairs to the base and fine-tuned models, following the LLM-as-a-Judge evaluation procedure. The answers generated by the models were then evaluated by GPT-3.5 Turbo, which acted as a judge. Additional details on this process can be found in Sec. 5.5.

The average win rate for $k = 3$ runs is shown in Fig. 6.1 as experiment **(a)** on the left side. The results suggest that in approx. 57% of the Q&A pairs, the answer of the fine-tuned model is preferred, while in approx. 36% of the pairs, the answer of the base model is preferred. In 5% of the pairs, the judge did not provide a valid answer, and in under 2% of the pairs, the judge decided on a Tie. The error bars, indicating the standard deviation of the k runs, are low for all possible outputs. That shows, on the one hand, that the LLM-as-a-Judge evaluation method is consistent over several runs, although the order of the models' answers will be randomized. On the other hand, fine-tuning the model improves the ability to answer questions regarding the Spyder IDE repository.

In addition to the results on all Q&A pairs, the model's performance on each dataset dimension gives exciting insights. The models performed best on the human-labelled dimension code semantics. With 62.38%, the model won almost two-thirds of the Q&A pairs. For the dependency dimension, the model was also better than the base model but had only a 54% win rate. The model performed the worst of all experiments in the meta-information dimension, indicating that the fine-tuning process from Self-Alignment reduced its performance in this dimension. The reasons are discussed in Sec. 6.7, where the results are categorized.

6.2.3 Results on Benchmarks

The following section presents the evaluation results of the fine-tuned model on the HumanEval [5] and MBPP [36] benchmarks. Fig. 6.2 presents the percentage of solved tasks by the base model Mistral 7B and the fine-tuned model. For each benchmark, the pass@1 and pass@10 are calculated. However, the results for both

Table 6.2: Average Win Rate in % for each dimension and experiment respectively on the SpyderChatQA. Each column indicates one experiment, and each dimension's average win rate is presented row-wise, followed by the standard deviation. Experiment **(a)** compares the finetuned Mistral 7B against Mistral 7B. **(b)** compares Mistral 7B with a RAG pipeline against Mistral 7B. **(c)** compares finetuned Mistral 7B with a RAG pipeline against Mistral 7B. **(d)** compares finetuned Mistral 7B against GPT 3.5. Standard deviation is calculated from $k = 3$ runs. Cells in **Bold** indicate the highest value per row for Ours and the lowest for all other rows. The cells underlined indicate the best value for all experiments with Mistral 7B as a base model.

	(a) Finetuned vs. Mistral	**(b) RAG vs. Mistral**	**(c) Combined vs. Mistral**	**(d) Combined vs. GPT 3.5**
	Code Semantics ($N = 140$)			
Ours	63.1% ± 3.2	62.38% ± 1.1	70.71% ± 3.5	**78.33% ± 3.8**
Base Model	27.86% ± 0.7	32.86% ± 0.7	25.24% ± 2.9	**16.19% ± 2.8**
No Value	7.38% ± 1.8	3.33% ± 1.1	3.81% ± 1.5	**4.76% ± 1.5**
Tie Bad	1.19% ± 0.4	0.71% ± 1.2	0.35% ± 0.5	0.71% ± 0.7
Tie Good	0.71% ± 1	0.71% ± 0.7	0% ± 0	0% ± 0
	Dependencies ($N = 135$)			
Ours	59.26% ± 2.56	54.07% ± 2.5	61.97% ± 1.9	**74.07% ± 1.5**
Base Model	35.56% ± 1.5	39.26% ± 1.9	33.1% ± 2.1	**17.29% ± 1.1**
No Value	**4.2% ± 2.3**	5.68% ± 0.8	**4.2% ± 0.4**	8.15% ± 1.3
Tie Bad	0.74% ± 1.3	0.49% ± 0.4	0.74% ± 1	0.25% ± 0.42
Tie Good	0.37% ± 0.5	0.49% ± 0.8	0.74% ± 0	0% ± 0
	Meta-Information ($N = 50$)			
Ours	38.67% ± 3.2	50.67% ± 1.1	**51.33% ± 3**	50.67% ± 5
Base Model	58.67% ± 6.1	42% ± 2	42.67% ± 2.3	**40.67% ± 4.2**
No Value	**2% ± 2**	6% ± 2	6% ± 2	7.33% ± 7.7
Tie Bad	0.67% ± 1.1	0% ± 0	0% ± 0	0% ± 0
Tie Good	0% ± 0	1.33% ± 1.1	0% ± 0	2% ± 2.8

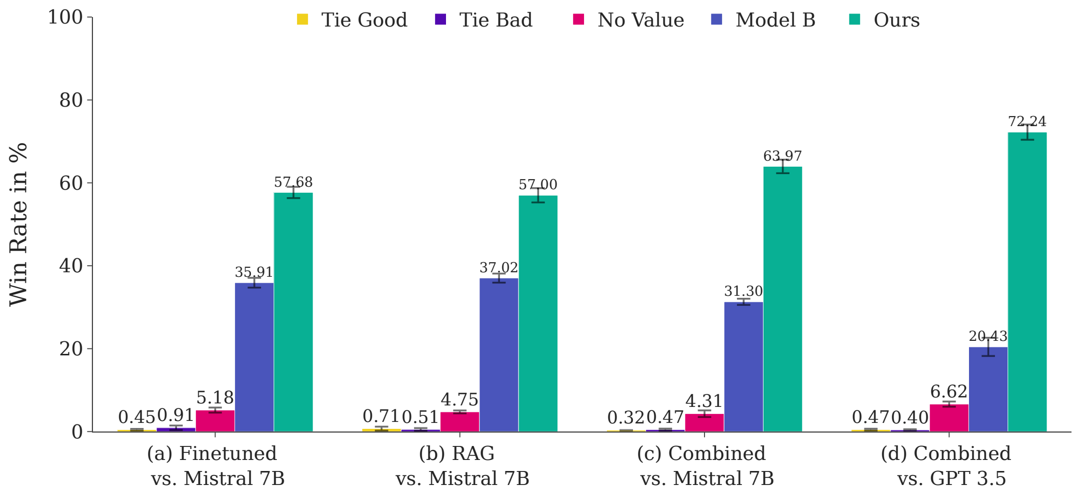

Figure 6.1: Average win rate for each experiment using LLM-as-a-Judge evaluation on the SpyderCodeQA. All experiments were executed with $k = 3$ where k is the number of runs. The error bars indicate the standard deviation of k runs. Experiment **(a)** compares the fine-tuned Mistral 7B against Mistral 7B. **(b)** compares Mistral 7B with a RAG pipeline against Mistral 7B. **(c)** compares fine-tuned Mistral 7B with a RAG pipeline against Mistral 7B. **(d)** compares finetuned Mistral 7B against GPT-3.5 Turbo.

benchmarks are pretty straightforward. The base model outperforms the fine-tuned model on HumanEval on pass@1 with 6.8% and on pass@10 with 8%. Similar results were found on the MBPP benchmark. Here, the effect is even higher, resulting in a difference of 11.5% on pass@1 and 12.6% on pass@10. These differences indicate that the general coding ability of the fine-tuned model has been reduced. One reason for the poorer performance could be the modified prompt template, as the model was not fine-tuned for pure coding tasks but for answering Q&A pairs.

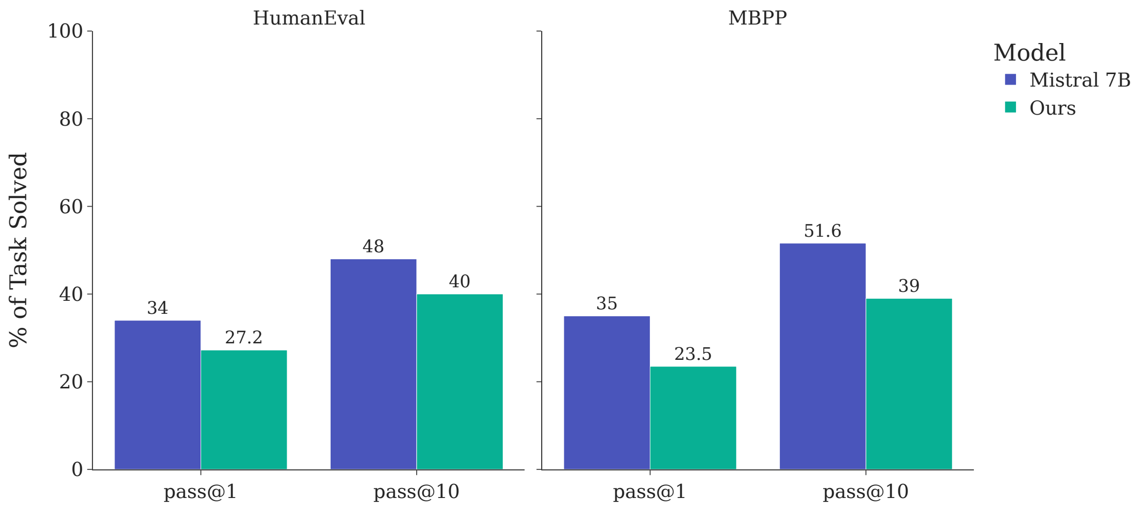

Figure 6.2: % of Tasked solved for HumanEval [5] & MBPP [36] for the base model Mistral 7B and the fine-tuned model without RAG.

6.3 RAG Pipeline

This section presents the results of the RAG pipeline. As a base model for the RAG pipeline, Mistral 7B Instruct v0.2 [22] was used.

6.3.1 Results on SpyderCodeQA

Each question of the SpyderCodeQA dataset was embedded, and $n = 1$ chunks were retrieved from the vector database for every Q&A pair. Then, each pair was passed once to the base model without context and once with the retrieved chunks as context, following the procedure of the LLM-as-a-Judge evaluation.

The average win rate for $k = 3$ runs is shown in Fig. 6.1 as the second from the left as experiment (b). For 57% of the Q&A pairs, the judge preferred Mistral 7B with the RAG pipeline, which aligns with the Self-Alignment pipeline, considering the standard deviation. Also, the win rate for the base model and the percentage of Q&A pairs that were not correctly judged are similar to the Self-Alignment pipeline.

The results of the different dataset dimensions differ from those of the Self-Alignment pipeline. Although both pipelines perform the same with a 1% difference in the code semantics dimension, there is a difference of 2 standard deviations in the results for the dependencies. The Meta-Information dimension shows the most significant difference, with the base model using the RAG pipeline outperforming the base model. This suggests that the RAG pipeline supports the model in answering questions related to the meta-information but is less valuable as supportive for answering questions regarding dependencies. This effect is discussed further in Section 6.7.

6.4 Self-Alignment + RAG

This section discusses the results of combining both pipelines to test whether they produce an interaction effect and outperform each pipeline. Therefore, the fine-tuned model was used in the RAG pipeline. Similar to the procedure of the RAG pipeline, each question of the SpyderCodeQA dataset was retrieved in a chunk of $n = 1$. The LLM-as-a-Judge evaluation is performed as described in Sec. 5.5. The fine-tuning and base models generated responses that were forwarded to the judge for final evaluation.

Mistral 7B The results for the comparison with Mistral 7B as the base model are shown in Fig. 6.1 as experiment (c), third from the left. Since this approach's average win rate is approximately 64%, which is higher than that of the two pipelines, respectively, this suggests an interaction effect between the two approaches. Also, the base model's average win rate is lower than each pipeline, respectively.

The results become even more interesting when examining each dimension separately. The best results were achieved for the code semantics dimension. With an average of 70% win rate, the model is in 7 out of 10 questions better than the base model. That indicates that this combination is a further improvement regarding code semantic questions. The results for the dependencies dimension are ambiguous. Although the average win rate of 61% is higher than the average, it indicates that the interaction of both pipelines improves the performance on the dependencies dimension. For the Meta-Information dimension, the results are disappointing. The model shows a 51% average win rate, which is no improvement over the base model if the standard deviation is considered. Reasons for that are discussed in Sec. 6.7.

GPT-3.5 Turbo In the last experiment, the Mistral 7B model was replaced with GPT-3.5 Turbo. It's worth noting that GPT-3.5 was utilized as the evaluator; therefore, it rated its own responses in this experiment. The specific model version used is `gpt-3.5-turbo-1106`, which has a context window of 16,385 tokens and can output a maximum of 4,096 tokens. The training data included data up to September 2021, and the model has not seen the Spyder IDE repository as training data. The results for the SpyderCodeQA are presented in Fig. 6.1 on the right side, referred to as experiment (d). The experiment's results were even more evident when compared to the experiments done with Mistral 7B. The combination of the fine-tuned model with an RAG pipeline outperformed GPT-3.5, with an average win rate of 72%. Only 20% of the Q&A pairs were won by GPT-3.5. However, it is worth noting that the rate of not finding a rating by the judge is slightly higher than with Mistral 7B.

Looking at the dimensions, the code semantics and dependencies results are even better, with 78.3% and 74.07%, respectively. That indicates that the fine-tuned model with the RAG pipeline is a better coding assistant than GPT-3.5. Only on the Meta-Information dimension does the model show a low performance, but that aligns with the results of Mistral 7B. With a standard deviation of 5, the difference between the two models is highly uncertain and should not be overinterpreted.

6.5 Ablation Studies

This section comprises additional experiments that aim to provide a deeper insight into the role of hyperparameters, such as the number of dataset samples, temperature, Top-P values, and the choice of judge, in influencing the results. The goal is to understand better how these hyperparameters affect the outcome and to provide a more comprehensive justification of the results.

6.5.1 Training Dataset Size

This section provides an overview of the investigation into the number of executions for the Self-Alignment approach, which was run twice for the main results pipeline. As said in Sec. 6.2.1, the first step of the Self-Augmentation is an executed n-times. The second step and the Self-Curation are always executed for all created Q&A pairs. Therefore, the Self-Augmentation was executed once (a), twice (b), and quadruple (c) to create these datasets. The related loss curves are shown in Appendix E in Fig. E.1 and the development of the learning rate in Fig. E.2. They were then used to train one model at a time, which was evaluated later on the SpyderCodeQA dataset. The RAG pipeline was utilized during the evaluation to compare them better with the main results. The results are shown in Fig 6.3, and the results for each dimension are attached in Appendix G on Tab. G.1.

In all three experiments, each fine-tuned model learned about the repository, as reflected in the higher average win rates compared to the base model. However, the best-performing model was achieved using the Self-Alignment pipeline twice to create the training dataset. The average win rate was considerably higher than the models trained with one or quadruple datasets, with an improvement of approximately three standard deviations. This difference is also visible regarding the win rate of the base model. However, there were only marginal differences in the number of 'No Values', Tie Good and Tie Bad compared to the other two models.

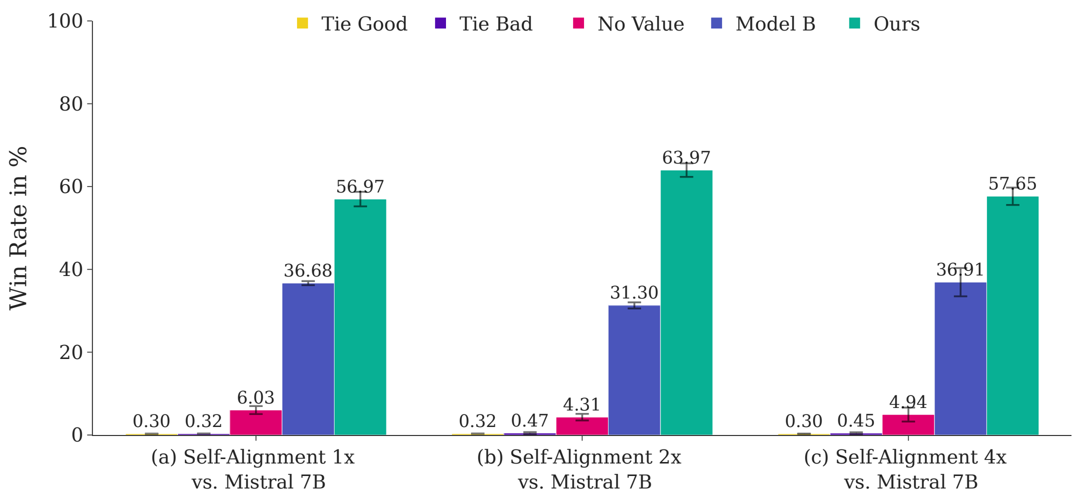

Figure 6.3: Average Win Rate in % for each experiment respectively on the SpyderCodeQA. Experiment **(a)** comparison of a fine-tuned model trained on the Self-Alignment pipeline **once** against Mistral 7B. **(b)** comparison of a fine-tuned model trained on the Self-Alignment pipeline **twice** against Mistral 7B. **(c)** comparison of a fine-tuned model trained on the Self-Alignment pipeline **quadruple** against Mistral 7B. Standard deviation is calculated from $k = 3$ runs.

Considering each dataset dimension, the results are ambiguous. On the one hand, the code semantic dimension has a high standard deviation for the twice and quadruple models, so no clear difference could be made while the average was higher. On the other hand, the difference was more evident for the dependencies dimension, with an average improvement of 8% in the win rate, indicating a significant improvement in the model's performance.

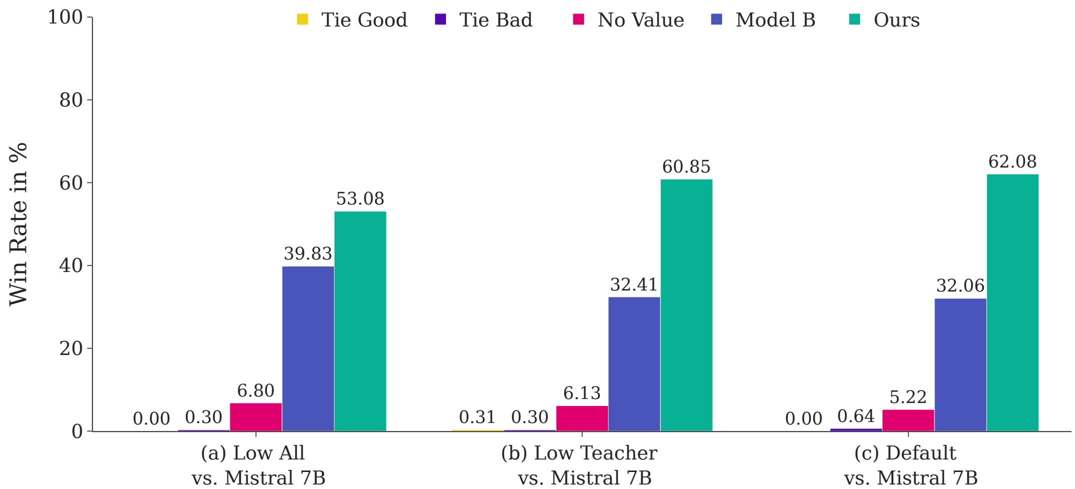

Figure 6.4: Average Win Rate in % for each experiment respectively on the Spyder-CodeQA. Experiment **(a)** Low temperature and Top-P for Teacher and Q&A dataset vs. Mistral 7B. **(b)** only reduced the temperature and Top-P for the Teacher dataset creation process vs. Mistral 7B. **(c)** Default parameters vs. Mistral 7B.

6.5.2 Hyperparameters for Data Augmentation

An experiment was conducted to observe the impact of varying temperature (t) and Top-P (p) parameters in the Self-Augmentation step. These hyperparameters control the behaviour of the model's responses during inference. The temperature parameter determines the level of randomness in the output generated by the model. A value of 0 will produce deterministic output, while a value of 2 will result in more random output. On the other hand, the Top-P parameter controls the number of possible words that are considered [114]. For instance, when setting the Top-P parameter to 0.9, the model will only consider the most probable words that account for 90% of the probability mass. The goal was to create a Teacher dataset D_0 and a Q&A dataset D_1 with different values for temperature and Top-P. Three setups were tested:

- **Low All** - Reduced temperature and Top-P values for both dataset creation steps D_0 and D_1 ($t = 0.3$, $p = 0.3$).
- **Low Teacher** - Reduced temperature and Top-P values only for creating the Teacher dataset D_0 ($t = 0.3$, $p = 0.3$) while keeping the Q&A dataset D_1 parameter default ($t = 0.7$, $p = 0.9$).

- **Default** - Default values for both dataset creation steps D_0 and D_1 ($t = 0.7$, $p = 0.9$).

A model was trained using SFT and QLoRA for each setup and evaluated with one run per setup with an RAG pipeline on the SpyderCodeQA. The experiment results are presented in Fig. 6.4. All three trained models in the experiment performed better than the base model but showed differences in their win rates. While the win rates for the Low Teacher and Default setups were the same, the Low All setup had a significantly lower win rate. This indicates that a high t and p when creating D_1 leads to higher performance on SpyderCodeQA afterwards. The assumption here is that the higher temperature and Top-P lead to more heterogeneous Q&A pairs generated from the generated output of the Teacher dataset D_0. A more diverse dataset will make the model more differentiated and robust. Moreover, since there was no significant difference between the Low Teacher and Default setups, it can be concluded that the temperature t and Top-P p used for creating the Teacher dataset D_0 do not significantly influence the performance on SpyderCodeQA.

6.5.3 Judgement with GPT-4 Turbo

This section explores the use of GPT-4 Turbo as a judge instead of GPT-3.5 Turbo, which was previously used in evaluating the models using the SpyderCodeQA dataset. However, using GPT-4 for this purpose is more expensive, and performing an LLM-as-a-Judge evaluation costs about \$5.60, which is 14 times more than GPT-3.5's cost of 40 cents. As a result, most of the evaluation in this thesis was done using GPT-3.5. However, to investigate whether there is any difference in the quality of judgment between the two models, both models were used to judge and evaluate the best-performing model using the RAG pipeline and the same hyperparameters on the SpyderCodeQA dataset. The results of this comparison are presented in Fig. 6.5, and the corresponding results for each dimension can be found in Appendix G, Tab. G.2.

The findings of this investigation provide exciting insights into the quality of the judgments. Both judges rated the quality of the response of the fine-tuned model with the RAG pipeline higher. However, GPT-4 preferred the fine-tuned model more than GPT-3.5. Moreover, the preference for the base model was even lower, as GPT-4 chose a tie in almost 10% as judgment, which is significantly more often than GPT-3.5. Furthermore, only 0.3% of the answers could not be assigned, indicating that GPT-4 can judge the performance of models more consistently and accurately.

Analyzing the individual dimensions presented in Tab. 6.5 is worth analyzing. Regarding code semantics, the fine-tuned model's win rate was quite similar, while the base model's win rate was 9% lower, making the difference between the two models even more significant. GPT-4 judged that approximately 8% of the questions were not correctly answered by both models, while 4.29% were equally good.

The results for the dependencies and meta-information dimension are even better. Both models showed an improvement of over 10% in win rate for the fine-tuned model. While in the judgement of GPT-3.5, the results for the meta-information showed no difference in win rate between the models, the assessment of GPT-4 gives hope that the fine-tuning model has also improved on this dimension. In total, GPT-4 confirms the results of the GPT-3.5 evaluation on SpyderCodeQA and strengthens them further since the results have improved.

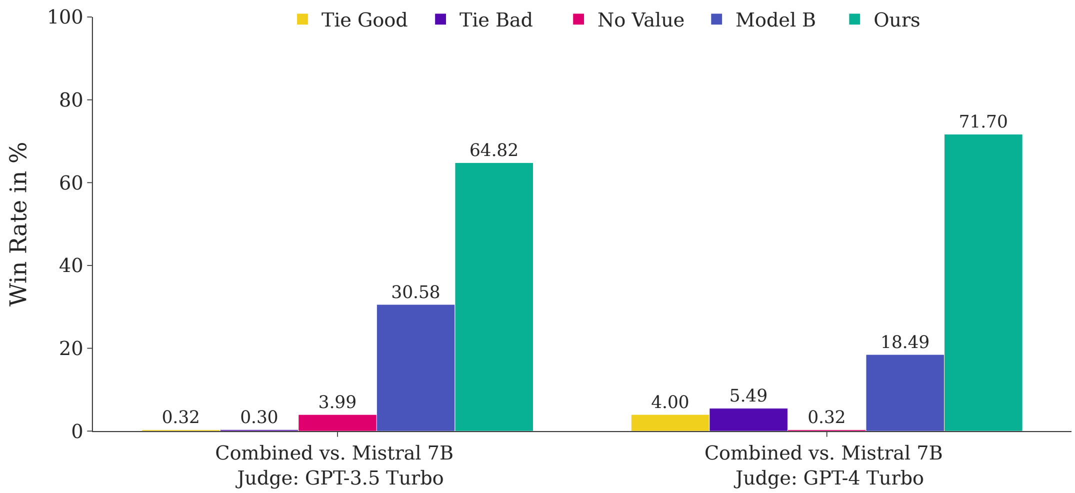

Figure 6.5: Average Win Rate in % for each experiment respectively on the SpyderCodeQA. **Left:** Fine-tuned model with RAG pipeline vs Mistral 7B judged by GPT-3.5 Turbo. **Right:** Fine-tuned model with RAG pipeline vs. Mistral 7B judged by GPT-4 Turbo.

6.6 Qualitative Analysis of SpyderCodeQA

This chapter presents an analysis based on the subset of Q&A pairs and the LLM-as-a-Judge evaluation to give more qualitative insights into the model performance and a deeper understanding of how fine-tuning and the use of RAG pipelines affect the output of the model. The examples are shown in Appendix F. Each example consists of the original question and answer, the answer of the two models and the judge's judgement at the end.

6.6.1 Code Semantics - Q&A pair 135

For Q&A pair 135 from SpyderCodeQA, the Self-Alignment and RAG pipeline evaluations are shown in Figs. F.1 and F.2, respectively. The answers and judgments for both combined are presented in Fig. F.3.

Focusing on the Self-Alignment evaluation in Fig. F.1, the question seeks an explanation of the class's functionality. The original answer, serving as a reference for the judge, provides a general description of the semantic purpose and the input and output values. The two models generate significantly different answers. As anticipated, the base model (Mistral 7B) states its inability to provide a precise answer due to lack of access to the code, attempting to infer the benefit from the name but remaining vague. Conversely, the fine-tuned model confidently explains the class's usage and returns a correct code snippet. However, comparing the generated code with the original reveals that the model hallucinates and does not provide the correct code. GPT-3.5 favours the fine-tuned model in its judgment despite the model's hallucination. The judge assumes the presented code snippet is correct and is satisfied with the answer, as it addresses the user's question and includes the code.

In comparison, the RAG pipeline evaluation in Fig. F.2 shows the provided context on the right side, indicating that the correct code snippet was fetched from the vector database because it contains the code of the requested class. While the base model's answer is nearly identical to the answer in the Self-Alignment evaluation, the base model fed with the needed code snippet gave a notable explanation of the class, explaining which superclass the model came from and its functionalities. Therefore, GPT-3.5's judgment again favours the modified variant, recognizing that the answer correctly explained the code's functionality.

Regarding the combination of both approaches in Fig. F.3, each approach demonstrates its benefits. While the fine-tuned model's answer was nicely formatted, it was contextually incorrect. In comparison, the RAG pipeline's answer was contextually correct but not formatted in a user-friendly manner. The combination of both approaches fulfilled both requirements, providing a well-formatted answer with a good explanation of the class and the correct code snippet. This example provides valuable insight into the effect each approach has on the base model. After training, the fine-tuned model changed the output format but could not reproduce the knowledge inserted through fine-tuning. It only improved its output format to become a better coding assistant. On the other hand, the RAG pipeline could insert the needed context but lacked an output adjustment. This underlines that combining both approaches leads to a better output, as the model benefits from both.

6.6.2 Code Semantics - Q&A pair 2

The evaluation results for Q&A pair 2 from SpyderCodeQA are presented in Fig. F.4 for the Fine-Tuned + RAG pipeline and in Fig. F.5 for the same model combination, but judged by GPT-4.

In this Q&A pair, the question seeks to explain the purpose of a test function within the test harness of the Cython client. The correct answer is that the function tests whether the Cython console is working correctly, which is a concise response. The base model's answer repeats information from the question and infers that the function's purpose is likely to test the Cython client. In contrast, the Fine-Tuned Model + RAG pipeline is provided with the relevant code snippet, including the code for the requested function, which enables the model to justify the purpose of the test function based on the interface comments. Not only does the model provide a more detailed explanation, but it also reproduces the code snippet. However, GPT-3.5 mistakenly prefers the base model's answer, failing to distinguish between the two responses and incorrectly crediting Assistant A with a precise description and code snippet.

In contrast, GPT-4's judgment is more diverse and accurate. By categorizing the answers into four dimensions - Relevance and Accuracy, Depth and Detail, Creativity, and Helpfulness - GPT-4's evaluation highlights the added value of the code snippet, which makes the answer more relevant, detailed, and creative. Consequently, GPT-4 correctly judges the Fine-Tuned Model's answer as superior. This example illustrates that evaluations assisted by LLMs can be tending to errors. However, leveraging larger and better-trained models makes the evaluation more granular and accurate, thereby minimizing errors.

6.6.3 Dependencies - Q&A pair 211

The evaluation results for Q&A pair 211 from SpyderCodeQA, which is part of the dependencies dimension, are presented in Figs. F.6, F.7, and F.8 for the Fine-Tuned model, RAG pipeline, and combined approaches, respectively.

This question is a typical example from the dependencies dimension, asking for all imported libraries in a file. The correct answer lists all imports used in the file. In evaluating the Fine-Tuned model (Fig. F.6), the base model correctly acknowledges its limitations, stating that without access to the file's source code, it cannot provide information about the imports used. In contrast, the Fine-Tuned model provides a confident but entirely incorrect answer, denying the presence of external library imports and listing non-existent internal imports from other repository modules. Unfortunately, the judgment fails to recognize the Fine-Tuned model's answer as incorrect, instead describing it as accurate.

In contrast, the answer provided by the base model with the RAG pipeline (Fig. F.7) is more accurate. When fed the correct file as context, the model lists all imports, including those from external libraries and internal repository modules. Although the response format is not aligned, the content is correct. This result is similar to that of Q&A 135. The combined evaluation of both approaches (Fig. F.8) generates an answer aligned to the provided format, listing only the external libraries without internal imports. This example demonstrates again that fine-tuning changes the output format of the model while the RAG pipeline effectively incorporates the correct context into the model.

6.6.4 Meta-Information - Q&A pair 313

The evaluation results for Q&A pair 313 from SpyderCodeQA, which falls under the meta-information dimension, are presented in Fig. F.9 and Fig. F.10. This question is a typical example from this dimension, as it inquires about the minimum size of the Spyder logo, a piece of information readily available in the README file or other documentation files, making it easily verifiable for correctness.

When evaluating the fine-tuned model, as shown in Figure F.9, it becomes apparent that the base model correctly indicates its lack of access to the source code or repository files. In contrast, the fine-tuned model attempts to guess the answer, incorrectly stating that the information is part of the README file. The judgement recognizes the fine-tuned model's answer as incorrect. Instead of penalizing both models for incorrect answers and judging Tie Bad, it favours the base model's response for its transparency in acknowledging the lack of information.

In contrast, the answers from the base and fine-tuned models using the RAG pipeline, presented in Figure F.10, demonstrate a more accurate response. The models' answers align perfectly with the correct answer by feeding the correct context into the prompt. The judgement also reflects a positive assessment of the RAG pipeline variant, as the answer conforms to the expected format and contains the correct information. This example illustrates the fine-tuned model's tendency to hallucinate without the context provided by the RAG pipeline, making it unable to answer the question about the repository. However, with the context, the model can effectively retrieve the relevant information and provide a correct answer.

6.6.5 Meta-Information - Q&A Pair 317

The evaluation results for Q&A pair 317 from the SpyderCodeQA dataset, which falls under the meta-information dimension, are shown in Fig. F.11. The question is typical for the meta-information category, asking how to install the Spyder Kernels using conda. The answer contains a natural language description and a shell command to install the spyder-kernels package. In this case, the base model provides a concise and comprehensive installation guide. It explains step-by-step how to create a conda environment, activate it, and set the necessary environment variables. In comparison, the fine-tuned model using the RAG pipeline provides the relevant context, including the installation markdown file for the spyder kernels. However, the model fails to generate a meaningful response. It initially produces repetitive Q&A pairs and then simply repeats the content of the markdown file. Additionally, the model cannot format the answer in natural language, instead retaining the markdown formatting. The evaluation suggests that the base model's answer provides a brief overview and a good explanation of the installation process. Therefore, the judge favours the answer from the base model. This example highlights that the model's response can still fail, even when provided with the proper context from the RAG pipeline. Despite having access to the relevant information, the model struggles to generate a coherent, natural language answer and properly format the output.

6.7 Summary

The results on the SpyderCodeQA evaluation dataset generated using the LLM-as-a-Judge evaluation show that fine-tuning and using an RAG pipeline improves the coding assistant's performance. That can be seen in Fig. 6.1, which shows that with 57%, the Fine-tuned model and the model using an RAG pipeline outperform the base model. In addition to that, the combination of both approaches shows an interaction effect because the win rate is approx. 64%, which is 7% higher than both approaches alone. Although the model showed promising results on the evaluation dataset, it experienced an average decrease of 9% on the HumanEval and MBPP benchmarks after being fine-tuned. Regarding the model's performance on different dataset dimensions, this thesis found that it performs exceptionally well for Code Semantics, which is the human-labeled dimension. While it also outperforms the base model on the dependencies dimension on GPT-3.5, the model performs even better when judged by GPT-4. However, the study's results indicate that the model's performance for Meta-Information is not very promising cause, most of the time, the performance between the fine-tuned and base models was equal.

The hyperparameter analysis demonstrates that the choice of the training dataset significantly impacts the model's performance. Doubling the training dataset seems to be the optimal way to achieve higher performance in the evaluation dataset. Utilizing high temperature and Top-P can lead to a more heterogeneous dataset in the Self-Alignment pipeline, which makes the model more differentiated and robust. In the last experiment, it was found that GPT-4 is a better judge than GPT-3.5. This is because GPT-4 has a lower Self-Alignment No Value rate and can decide on Tie more often. Additionally, there was a significant improvement in the performance of the meta-information dimension when GPT-4 was used as a judge. One possible reason for this improvement is that the correct answers of the Q&A pairs are significantly longer and consist of more sentences than the other two dimensions, as shown in the analysis of the evaluation dataset in Sec. 4.4.2. Due to this more extended context, the judgment of the answers could be worse because GPT-3.5 cannot work with such an extended context and has difficulties producing valid judgments.

About the examples presented in Sec. 6.6, it can be concluded that the Fine-tuning process alters the output format of the model but may occasionally produce inaccurate content. Conversely, the RAG pipeline typically generates accurate content but may not always display the appropriate output format. When both methods are combined, as demonstrated in the evaluated examples, the model exhibits an interaction effect, effectively adjusting the output format while maintaining correct content in most cases.

7. Discussion

In the final chapter, the research results are summarized and discussed. First, the work is concluded in Sec. 7.1. This is followed by a categorization of each research question in Secs. 7.2, 7.3 and 7.4. Each research question is structured the same way. It starts with a short overview of what was done and how the question can be answered based on the evaluations conducted. This is followed by implications resulting from the answer, an overview of known limitations, and emerging future work.

7.1 Conclusion

This paper presents a set of experiments to evaluate the performance of LLMs at the repository level Q&A. The experiments were designed to address three research questions: (**RQ1**) the impact of fine-tuning an LLM with self-augmented data on model performance, (**RQ2**) the effect of an RAG pipeline on repository-level code question answering, and (**RQ3**) the interaction effect of combining a fine-tuned model with an RAG pipeline. A novel approach inspired by Li et al. [29] was employed to create a training dataset through Self-Alignment. This approach utilizes multiple prompt templates to extract information from repository chunks of the Spyder IDE, generating Q&A dataset pairs for SFT [30] training using QLoRA [31]. The RAG pipeline was constructed by embedding the same repository chunks using Instructor [35] and storing them in a Chroma Vector database[1].

Addressing the research questions required a suitable dataset representative of a repository to test the abilities of the fine-tuned LLM or LLMs with RAG pipelines. Due to the lack of an appropriate dataset in the literature, SpyderCodeQA was created. This dataset consists of 325 Q&A pairs related to the Spyder IDE[2] repository, divided into three dimensions: code semantics, dependency understanding, and meta-information understanding. These dimensions assess the ability of LLMs to answer questions at the repository level. The Q&A pairs were used to evaluate the approaches by comparing them with the base model using LLM-as-a-Judge evaluation. Additionally, the HumanEval [5] and MBPP [36] benchmarks were conducted to verify the general coding generation abilities. The experimental results suggest that fine-tuning and RAG pipelines improve the abilities of coding assistants

[1] https://www.trychroma.com

[2] https://github.com/spyder-ide/spyder

to significantly outperform the base model on the repository-level coding Q&A with approx. 20% and show an interaction effect when combining both approaches that further improves the performance on SpyderCodeQA. However, the benchmark results indicate a decline in code generation skills, with an average decrease of 9.65% in the pass@1 rate.

7.2 Research Question 1 (RQ1)

The first research question investigated whether self-augmented data improves the performance of answering code questions at the repository level. The Spyder IDE repository was utilized to answer the research question. The repository was partitioned into chunks and then processed through the Self-Alignment pipeline. This process generated a Q&A dataset further used as training data. Henceforward, the base model Mistral 7B [22] was fine-tuned with QLoRA.

The fine-tuned model's performance was evaluated using LLM-as-a-Judge on SpyderCodeQA, HumanEval [5], and MBPP [36] benchmarks. The results indicate that the fine-tuned model, trained on self-augmented data, improves coding-related Q&A, with an average win rate approximately 20% higher than the base model, demonstrating clear improvement.

With regard to the SpyderCodeQA evaluation, the research question can be answered affirmatively. Specifically, the model shows a significant improvement in the Code Semantics dimension, the only human-labelled dimension, suggesting that the model can align with user requests and fetch the necessary context incorporated during SFT into the model weights. The dependency dimension also exhibited improvement after fine-tuning, indicating that inserting imports as the context in the training dataset aids the model in understanding the repository's structure. However, the model's performance in the meta-information dimension decreased, possibly due to difficulties fetching the correct information after fine-tuning. Further investigation is required to explain this phenomenon fully.

The benchmarks show contrasting results to those of SpyderCodeQA. On both Benchmarks, the model performance decreased significantly after the fine-tuning. One reason for that can be that the model catastrophically forgot its code generation abilities. Another possible explanation is that the model struggles to generate the code in the right format through the Alignment of the Q&A format of the self-augmented dataset. The training data often contained explanations of the code's functionality rather than the code itself. A possible solution for that problem would be to change the Self-Alignment pipeline to create a more diverse dataset that includes Q&A pairs mainly focused on code generation.

The results of the qualitative analysis of the examples in Sec. 6.6 indicate that the model displayed an adapted response format more aligned with user interaction. Additionally, the model often could not replicate the knowledge from the training dataset for fine-tuning.

Implications These results have further implications for possible usage and future work. After fine-tuning the model with the training dataset created through the Self-Alignment process. This effect suggests that adjusting the prompt templates of the Self-Alignment process changes the way the model behaves. For example, the questions (Sec. C) that were used in the pipeline (Shown in Fig. 5.2) to create the Teacher Dataset D_0 could be changed for further alignment to actual user requests. Another possible improvement is generalising the prompt templates for the Teacher dataset D_0 and Q&A dataset D_1, as they are currently tailored to one specific Python repository.

Another implication is that the training data created in the Self-Alignment pipeline does not contribute knowledge to the model. To train the output format, the repository does not necessarily have to be used for learning; instead, the focus should be on defining specified output formats for frameworks, programming languages, or projects, which can then be applied to the models via fine-tuning.

Limitations However, despite the attempt to design the experiments as validly and objectively as possible, essential limitations must be considered when interpreting the results. The Q&A pairs generated by the Self-Alignment process may not be semantically and syntactically correct. Although the model has been trained to match questions with the corresponding answers, it is not guaranteed that the generated code is functionally correctly reproduced and that the generated question is similar to a user request. The model itself curates the Q&A pairs, but the curation can only verify if the question matches the answer and seems to be correct. Therefore, this pipeline step could further improve the curation/verification process.

In Sec. 4.5, the limitations of the evaluation dataset were discussed. It is worth noting that the evaluation is limited to one Python repository that has its own unique structure. This is important to consider as the model may behave differently when applied to other repositories, which could result in biased results. In addition, the evaluation results only cover a limited set of questions that could arise in relation to repositories. Given the wide range of programming languages, frameworks, and projects, these results may not apply to all scenarios.

Regarding evaluating the model's performance, the LLM-as-a-Judge approach also has limitations. Despite the elimination of the position bias and the attempts to use GPT-4 as a judge, the evaluation is not flawless. The superior model judges the answers, but sometimes, the criteria are chosen by the model itself and do not match those of humans. Also, the correctness of the produced code is often not sufficiently verifiable for the model, as it does not have access to the necessary source code.

Future Work Several future works have resulted from the insights of this study's implications and elaborated limitations. One promising direction is to enhance the Self-Alignment approach by ensuring the correctness of generated training samples and creating more diverse Q&A pairs. This could lead to a more robust model capable of handling various programming languages and repositories. Additionally, fully fine-tuning the model with the training dataset could allow it to query information more effectively, and exploring this possibility could yield valuable insights.

To better align the model with user needs, human evaluation is essential. It would provide additional insights since humans are the target audience for Q&A on repository-level programming, and they often have more knowledge about the repository, allowing them to judge the model's responses better. Regarding the LLM-as-a-Judge evaluation, the qualitative analysis suggests that GPT-4 Turbo is a more objective judge, capable of effectively differentiating answers. Therefore, it is recommended to use GPT-4 Turbo in all future analyses.

7.3 Research Question 2 (RQ2)

The second research question investigated if an RAG pipeline improves the performance of repository-level code Q&A. The RAG pipeline generates a vector representation of the repository data using chunks (as described in Sec. 5.1), which are then stored in a vector database. When queried, the relevant chunks are retrieved and passed as context to the model. The implementation of the RAG pipeline is illustrated in Sec. 5.4.

With regard to the evaluation of SpyderCodeQA, the research question can be answered affirmatively. Similar to the results of **RQ1**, the model significantly improves the Code Semantics dimension, the only human-labelled dimension. This indicates that the information in the prompt is suitable, ensuring that the model receives the correct context to answer the question correctly. However, on the dependency dimension, the model's performance is slightly inferior to the performance of the Self-Alignment approach. The model seems to use only the RAG pipeline and faces difficulties in aligning its answers with the user's output format, as discussed in

Sec. 6.6.3. Although it often provides the correct information, it struggles to reduce the volume of information and answer the question precisely. This is also consistent with the results of Q&A 317 (Sec. 6.6.5), which is part of the Meta-Information dimension. In this Q&A pair, the model has problems aligning to the correct format and has difficulties processing the given context to extract the correct information and generate a meaningful answer.

The other Q&A pairs presented in Sec. 6.6 indicate that, in most cases, the model is provided with the correct context (Shown in Figs. F.2, F.7 & F.10), which suggests that the Instructor embedding model generates meaningful embeddings to locate the appropriate chunk in the vector database. Also, the model seems to utilize the context to answer the question, particularly for the code semantics dimension. This shows that the chunk size and overlap are appropriately chosen to provide sufficient context. However, additional experimentation is required for other dimensions to identify issues in generating responses.

Furthermore, a significant advantage of the RAG pipeline over Self-Alignment is that the weights of the base model remain unchanged, ensuring that the benchmark performance remains unaffected.

Implications The findings indicate that using RAG is a suitable and effective method for enhancing the model's knowledge about the repository. However, several factors determine the performance of the model. It is crucial that the model utilizes the provided context and selects the proper output format to generate an accurate answer. Other factors, such as the embedding model and the chunk size, can also influence whether the model provides the correct and adequate information.

It is essential to consider certain factors to build a successful RAG pipeline for repository-level Q&A. First, the knowledge transfer into the model through an RAG pipeline works better than the Self-Alignment approach, given this specific training setup and data set. Second, the RAG pipeline is easier to maintain and update and only requires GPUs for inference, not training.

Limitations Despite its advantages, the RAG pipeline has some limitations that must be considered. One major limitation is that the context provided to the LLM is always just a portion of the file, which means that knowledge about multiple files is not processed. The connection between the files and the code cannot be considered. To address this, the context would need to be preprocessed better. One possible solution is to have a hierarchical structure that provides context at different levels and contains summarized knowledge. For example, a description of what a module is responsible for or how its general structure could be added to each chunk of each file in the module. That additional information should help the model gain a deeper understanding of the repository.

Another limitation is the number of chunks retrieved using the RAG pipeline. For all experiments, the number of chunks was set to $N = 1$, but it could also be interesting to test whether the number of chunks could further improve the model's performance. Also, the size of the chunk and the overlapping characters are possible variables for optimizing the results.

Future Work Several potential optimizations for the RAG pipelines could be explored in the future. Additionally, testing the number of retrieved documents could help further optimize the model's ability to retrieve relevant information. As described in the limitations, preprocessing the chunks and adding hierarchy knowledge from superior modules shows excellent potential to provide answers to a single code file and create a coding assistant with superior knowledge about the repository. Furthermore, pre- and post-retrieval optimization techniques, as suggested by Gao et al. [66], could also be implemented to enhance the performance of the RAG pipeline further. These techniques involve optimizing the retrieval and summarization of information from the input data, which could improve the model's accuracy and efficiency.

7.4 Research Question 3 (RQ3)

The third research question investigated the interaction effect of combining a fine-tuned model and an RAG pipeline on performance. Both approaches, described in Sec. 5.2 and Sec. 5.3, were applied to the Spyder IDE repository, and the evaluation was conducted using the SpyderCodeQA dataset.

The SpyderCodeQA evaluation confirms that both RQ1 and RQ2 have been answered positively, and the focus now lies on determining if the effects show an interaction. As shown in Fig. 6.1 for all Q&A pairs and in Tab. 6.2 for each dimension, the combination of both approaches is better than each approach individually. With approx. 7% higher average win rate than both approaches individually and a lower win rate of the base model, the effect is sufficient. This result is supported by the fact that for each of the three dimensions, there is an increase in the win rate and a reduction in the base model win rate. In addition, the results against GPT-3.5 also show that the combination is superior to a strong LLM. Also, the experiments using GPT-4 Turbo as Judge shown in Sec. 6.5 suggest even further that the combination of both approaches outperforms the base model and is a promising approach for repository-level Q&A.

The qualitative analysis in Sec. 6.6 provides meaningful insights into how the interaction effect. The Q&A example 135 in Fig. F.3 is an excellent example of how the output format is adjusted from the fine-tuning process while the proper knowledge, e.g. context, is provided through the RAG pipeline. The same applies to Q&A example 211 in Fig. F.8 for the Dependency dimension and also for the Meta-Information dimension in Q&A 313 in Fig. F.10.

Implications The findings suggest that combining both approaches can be a suitable strategy for aligning the format through fine-tuning and providing knowledge through the RAG pipeline. Specifically, fine-tuning with QloRA did not produce any noticeable effect on knowledge transfer, while vice versa, the RAG pipeline did not affect changing the output format. It is recommended that both methods be explored to gain better insights into their respective benefits and limitations to further develop a coding assistant.

Limitations The limitations sections of both approaches have highlighted several constraints. Still, a significant challenge in the interaction of both approaches is the impact of the fine-tuning process on the effective and accurate use of context. As mentioned in Sec. 6.2, the model's performance decreased on both MBPP [36] and HumanEval [5] benchmarks following the fine-tuning process. This shows that fine-tuning can change models' abilities to perform specific tasks. Therefore, the experiments that were conducted do not clarify how the model enhances its capacity to handle the context, particularly source code, and grasp it deeper after the fine-tuning process.

Future Work In terms of future work, many of the ideas for both approaches can apply to the interaction effect. However, a new concept to explore is possibly adapting the model to a specific output format. This output format can be generated from the question using pre-processing techniques to rewrite the prompt, as suggested in the survey conducted by [66]. The newly generated rewritten prompt is optimized for similarity search in the vector database, thus enhancing the retrieval accuracy.

Bibliography

[1] Hugo Touvron et al. "Llama 2: Open Foundation and Fine-Tuned Chat Models". In: *CoRR* abs/2307.09288 (2023). arXiv: 2307.09288. DOI: 10.48550/ARXIV.2307.09288.

[2] Mike Lewis et al. "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 2020, pp. 7871–7880.

[3] Tingwei Zhu et al. "Deep is better? An empirical comparison of information retrieval and deep learning approaches to code summarization". In: *ACM Transactions on Software Engineering and Methodology* 33.3 (2024). Publisher: ACM New York, NY, pp. 1–37.

[4] Erik Nijkamp et al. "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis". In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL: https://openreview.net/pdf?id=iaYcJKpY2B%5C_.

[5] Mark Chen et al. "Evaluating Large Language Models Trained on Code". In: *arXiv e-prints* (2021), arXiv–2107.

[6] D. Nam et al. "Using an LLM to Help With Code Understanding". In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. ISSN: 1558-1225. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2024, pp. 881–881.

[7] Qinyu Luo et al. "RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation". In: *arXiv e-prints* (2024), arXiv–2402.

[8] Raymond Li et al. "StarCoder: may the source be with you!" In: *arXiv e-prints* (2023), arXiv–2305.

[9] Nhan Nguyen and Sarah Nadi. "An empirical evaluation of GitHub copilot's code suggestions". In: *Proceedings of the 19th International Conference on Mining Software Repositories*. Pittsburgh Pennsylvania, May 2022, pp. 1–5. URL: https://dl.acm.org/doi/10.1145/3524842.3528470.

[10] Liang Niu et al. "CodexLeaks: Privacy Leaks from Code Generation Language Models in GitHub Copilot". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Aug. 2023, pp. 2133–2150. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/niu.

[11] Hugo Touvron et al. "LLaMA: Open and Efficient Foundation Language Models". In: *CoRR* abs/2302.13971 (2023). arXiv: 2302.13971. DOI: 10.48550/ARXIV.2302.13971.

[12] Stephan Schunkert et al. *Der UBA-CO2-Rechner für Privatpersonen:* 2022.

[13] Patrick Lewis et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. 2020, pp. 9459–9474.

[14] Omar Khattab and Matei Zaharia. "ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT". In: *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 39–48. URL: https://doi.org/10.1145/3397271.3401075.

[15] Keshav Santhanam et al. "PLAID: An Efficient Engine for Late Interaction Retrieval". In: *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. CIKM '22. event-place: Atlanta, GA, USA. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1747–1756. URL: https://doi.org/10.1145/3511808.3557325.

[16] Kurt Shuster et al. "Retrieval Augmentation Reduces Hallucination in Conversation". In: *arXiv e-prints* (2021), arXiv–2104.

[17] Md Rizwan Parvez et al. "Retrieval Augmented Code Generation and Summarization". In: *Findings of the Association for Computational Linguistics: EMNLP 2021*. 2021, pp. 2719–2734.

[18] Daoguang Zan et al. "When Language Model Meets Private Library". In: *Findings of the Association for Computational Linguistics: EMNLP 2022*. 2022, pp. 277–288.

[19] Shuyan Zhou et al. "Docprompting: Generating code by retrieving the docs". In: *The Eleventh International Conference on Learning Representations*. 2022.

[20] Neil Houlsby et al. "Parameter-efficient transfer learning for NLP". In: *International Conference on Machine Learning*. 2019, pp. 2790–2799.

[21] Andreas Köpf et al. “OpenAssistant Conversations–Democratizing Large Language Model Alignment”. In: *arXiv e-prints* (2023), arXiv–2304.

[22] Albert Q Jiang et al. “Mistral 7B”. In: *arXiv e-prints* (2023), arXiv–2310.

[23] Yang Liu et al. “GPTEval: NLG Evaluation using GPT-4 with Better Human Alignment”. In: *arXiv e-prints* (2023), arXiv–2303.

[24] Xiao Liu et al. “P-Tuning v2: Prompt Tuning Can Be Comparable to Fine-tuning Universally Across Scales and Tasks”. In: *arXiv e-prints* (2021), arXiv–2110.

[25] Edward J Hu et al. “LoRA: Low-Rank Adaptation of Large Language Models”. In: *arXiv e-prints* (2021), arXiv–2106.

[26] Renrui Zhang et al. “LLaMA-Adapter: Efficient Fine-tuning of Language Models with Zero-init Attention”. In: *arXiv e-prints* (2023), arXiv–2303.

[27] Yunxiang Li et al. “ChatDoctor: A Medical Chat Model Fine-Tuned on a Large Language Model Meta-AI (LLaMA) Using Medical Domain Knowledge”. In: *Cureus* (June 2023). URL: https://www.cureus.com/articles/152858-chatdoctor-a-medical-chat-model-fine-tuned-on-a-large-language-model-meta-ai-llama-using-medical-domain-knowledge.

[28] Tiedong Liu and Bryan Kian Hsiang Low. “Goat: Fine-tuned LLaMA Outperforms GPT-4 on Arithmetic Tasks”. In: *arXiv e-prints* (2023), arXiv–2305.

[29] Xian Li et al. “Self-Alignment with Instruction Backtranslation”. In: *arXiv e-prints* (2023), arXiv–2308.

[30] Long Ouyang et al. “Training language models to follow instructions with human feedback”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 27730–27744.

[31] Tim Dettmers et al. “QLoRA: Efficient Finetuning of Quantized LLMs”. In: *arXiv e-prints* (2023), arXiv–2305.

[32] Arghavan Moradi Dakhel et al. “GitHub Copilot AI pair programmer: Asset or Liability?” In: *Journal of Systems and Software* 203 (2023), p. 111734. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2023.111734.

[33] Sida Peng et al. “The Impact of AI on Developer Productivity: Evidence from GitHub Copilot”. In: *arXiv e-prints* (2023), arXiv–2302.

[34] Hamel Husain et al. “CodeSearchNet Challenge: Evaluating the State of Semantic Code Search”. In: *arXiv e-prints* (2020).

[35] Hongjin Su et al. "One Embedder, Any Task: Instruction-Finetuned Text Embeddings". In: *arXiv e-prints* (2022).

[36] Jacob Austin et al. "Program Synthesis with Large Language Models". In: *arXiv e-prints* (2021), arXiv–2108.

[37] Lianmin Zheng et al. "Judging LLM-as-a-judge with MT-Bench and Chatbot Arena". In: *arXiv e-prints* (2023).

[38] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* (2017).

[39] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[40] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning Internal Representations by Error Propagation, Parallel Distributed Processing, Explorations in the Microstructure of Cognition, ed. DE Rumelhart and J. McClelland. Vol. 1. 1986". In: *Biometrika* 71 (1986), pp. 599–607.

[41] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. "A neural probabilistic language model". In: *Advances in neural information processing systems* 13 (2000).

[42] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997). Publisher: MIT press, pp. 1735–1780.

[43] Alex Graves. "Generating Sequences With Recurrent Neural Networks". In: *arXiv e-prints* (2013), arXiv–1308.

[44] Tomás Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings.* Ed. by Yoshua Bengio and Yann LeCun. 2013. URL: http://arxiv.org/abs/1301.3781.

[45] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems* 27 (2014).

[46] Yonghui Wu et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: *CoRR* abs/1609.08144 (2016). arXiv: 1609.08144. URL: http://arxiv.org/abs/1609.08144.

[47] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1409.0473.

[48] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of NAACL-HLT*. 2019, pp. 4171–4186.

[49] Yinhan Liu et al. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *CoRR* abs/1907.11692 (2019). arXiv: 1907.11692. URL: http://arxiv.org/abs/1907.11692.

[50] Victor Sanh et al. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: *CoRR* abs/1910.01108 (2019). arXiv: 1910.01108. URL: http://arxiv.org/abs/1910.01108.

[51] Zhenzhong Lan et al. "ALBERT: A Lite BERT for Self-supervised Learning of Language Representations". In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=H1eA7AEtvS.

[52] Alec Radford et al. "Improving language understanding by generative pre-training". In: (2018). Publisher: OpenAI.

[53] Alec Radford et al. "Language models are unsupervised multitask learners". In: *OpenAI blog* 1.8 (2019), p. 9.

[54] Tom Brown et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[55] Wayne Xin Zhao et al. "A Survey of Large Language Models". In: *arXiv e-prints* (2023), arXiv–2303.

[56] SS Manathunga and YA Illangasekara. "Retrieval Augmented Generation and Representative Vector Summarization for large unstructured textual data in Medical Education". In: *arXiv e-prints* (2023).

[57] Jiaxi Cui et al. "ChatLaw: Open-Source Legal Large Language Model with Integrated External Knowledge Bases". In: *arXiv e-prints* (2023).

[58] Colin Raffel et al. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". In: *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67. URL: http://jmlr.org/papers/v21/20-074.html.

[59] Yizhong Wang et al. "Self-Instruct: Aligning Language Models with Self-Generated Instructions". In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 13484–13508. DOI: 10.18653/v1/2023.acl-long.754.

[60] Swaroop Mishra et al. "Cross-Task Generalization via Natural Language Crowdsourcing Instructions". In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*. Ed. by Smaranda Muresan, Preslav Nakov, and Aline Villavicencio. Association for Computational Linguistics, 2022, pp. 3470–3487. DOI: 10.18653/V1/2022.ACL-LONG.244.

[61] Sahil Chaudhary. *Code Alpaca: An Instruction-following LLaMA model for code generation*. Publication Title: GitHub repository. 2023. URL: https://github.com/sahil280114/codealpaca.

[62] Xiang Lisa Li and Percy Liang. "Prefix-Tuning: Optimizing Continuous Prompts for Generation". In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*. Ed. by Chengqing Zong et al. Association for Computational Linguistics, 2021, pp. 4582–4597. DOI: 10.18653/V1/2021.ACL-LONG.353.

[63] Paul F. Christiano et al. "Deep Reinforcement Learning from Human Preferences". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon et al. 2017, pp. 4299–4307. URL: https://proceedings.neurips.cc/paper/2017/hash/d5e2c0adad503c91f91df240d0cd4e49-Abstract.html.

[64] Dylan Hadfield-Menell et al. "The Off-Switch Game". In: *The Workshops of the The Thirty-First AAAI Conference on Artificial Intelligence, Saturday, February 4-9, 2017, San Francisco, California, USA*. Vol. WS-17. AAAI Technical Report. AAAI Press, 2017. URL: http://aaai.org/ocs/index.php/WS/AAAIW17/paper/view/15156.

[65] Jeff Wu et al. "Recursively Summarizing Books with Human Feedback". In: *CoRR* abs/2109.10862 (2021). arXiv: 2109.10862. URL: https://arxiv.org/abs/2109.10862.

[66] Yunfan Gao et al. "Retrieval-Augmented Generation for Large Language Models: A Survey". In: *arXiv e-prints* (2023), arXiv–2312.

[67] Kelvin Guu et al. "Retrieval Augmented Language Model Pre-Training". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. July 2020, pp. 3929–3938. URL: https://proceedings.mlr.press/v119/guu20a.html.

[68] Xiang Ling et al. "Deep Graph Matching and Searching for Semantic Code Retrieval". In: *ACM Transactions on Knowledge Discovery from Data* 15.5 (Oct. 2021), pp. 1–21. URL: https://dl.acm.org/doi/10.1145/3447571.

[69] Xianming Li and Jing Li. "AnglE-optimized Text Embeddings". In: *CoRR* abs/2309.12871 (2023). arXiv: 2309.12871. DOI: 10.48550/ARXIV.2309.12871.

[70] Zhuyun Dai et al. "Promptagator: Few-shot Dense Retrieval From 8 Examples". In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL: https://openreview.net/pdf?id=gmL46YMpu2J.

[71] Baptiste Rozière et al. "Code Llama: Open Foundation Models for Code". In: *arXiv e-prints* (2023), arXiv–2308.

[72] Steven I Ross et al. "The programmer's assistant: Conversational interaction with a large language model for software development". In: *Proceedings of the 28th International Conference on Intelligent User Interfaces*. 2023, pp. 491–514.

[73] Vincent J Hellendoorn and Premkumar Devanbu. "Are deep neural networks the best choice for modeling source code?" In: *Proceedings of the 2017 11th Joint meeting on foundations of software engineering*. 2017, pp. 763–773.

[74] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. "Repository-level prompt generation for large language models of code". In: *International Conference on Machine Learning*. PMLR, 2023, pp. 31693–31715.

[75] Fengji Zhang et al. "RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 2023, pp. 2471–2484.

[76] Ramakrishna Bairi et al. "CodePlan: Repository-level Coding using LLMs and Planning". In: *CoRR* abs/2309.12499 (2023). arXiv: 2309.12499. DOI: 10.48550/ARXIV.2309.12499.

[77] Noor Nashid, Mifta Sintaha, and Ali Mesbah. "Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2023, pp. 2450–2462.

[78] Runchu Tian et al. "DebugBench: Evaluating Debugging Capability of Large Language Models". In: *arXiv e-prints* (2024), arXiv–2401.

[79] Martin Weyssow et al. "Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models". In: *arXiv e-prints* (2023), arXiv–2308.

[80] Zhiqiang Yuan et al. "Evaluating Instruction-Tuned Large Language Models on Code Comprehension and Generation". In: *arXiv e-prints* (2023), arXiv–2308.

[81] Oded Ovadia et al. "Fine-Tuning or Retrieval? Comparing Knowledge Injection in LLMs". In: *arXiv e-prints* (2023), arXiv–2312.

[82] Heydar Soudani, Evangelos Kanoulas, and Faegheh Hasibi. "Fine Tuning vs. Retrieval Augmented Generation for Less Popular Knowledge". In: *arXiv e-prints* (2024), arXiv–2403.

[83] Bosheng Ding et al. "Data Augmentation using LLMs: Data Perspectives, Learning Paradigms and Challenges". In: *arXiv e-prints* (2024), arXiv–2403.

[84] Baolin Peng et al. "Instruction Tuning with GPT-4". In: *arXiv e-prints* (2023).

[85] Zeming Dong et al. "Boosting Source Code Learning with Data Augmentation: An Empirical Study". In: *arXiv e-prints* (2023), arXiv–2303.

[86] Kechi Zhang et al. "ToolCoder: Teach Code Generation Models to use APIs with search tools". In: *arXiv e-prints* (2023), arXiv–2305.

[87] Shishir G Patil et al. "Gorilla: Large Language Model Connected with Massive APIs". In: *arXiv e-prints* (2023), arXiv–2305.

[88] Chenxiao Liu and Xiaojun Wan. "CodeQA: A Question Answering Dataset for Source Code Comprehension". In: *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*. Ed. by Marie-Francine Moens et al. Association for Computational Linguistics, 2021, pp. 2618–2632. DOI: 10.18653/V1/2021.FINDINGS-EMNLP.223.

[89] Junjie Huang et al. "CoSQA: 20,000+ Web Queries for Code Search and Question Answering". In: *arXiv e-prints* (2021), arXiv–2105.

[90] Changyoon Lee, Yeon Seonwoo, and Alice Oh. "CS1QA: A Dataset for Assisting Code-based Question Answering in an Introductory Programming Course". In: *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2022, Seattle, WA, United States, July 10-15, 2022*. Ed. by Marine Carpuat, Marie-Catherine de Marneffe, and Iván Vladimir Meza Ruíz. Association for Computational Linguistics, 2022, pp. 2026–2040. DOI: 10.18653/V1/2022.NAACL-MAIN.148.

[91] Zehan Li et al. "ProCQA: A Large-scale Community-based Programming Question Answering Dataset for Code Search". In: *arXiv e-prints* (2024), arXiv–2403.

[92] Kishore Papineni et al. "Bleu: a method for automatic evaluation of machine translation". In: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, pp. 311–318.

[93] Chin-Yew Lin and FJ Och. "Looking for a few good metrics: ROUGE and its evaluation". In: *Ntcir workshop*. 2004.

[94] Yupeng Chang et al. "A Survey on Evaluation of Large Language Models". In: *arXiv e-prints* (2023), arXiv–2307.

[95] Qinkai Zheng et al. "CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X". In: *arXiv e-prints* (2023), arXiv–2303.

[96] Yuhang Lai et al. "DS-1000: A natural and reliable benchmark for data science code generation". In: *International Conference on Machine Learning*. PMLR, 2023, pp. 18319–18345.

[97] Peter Clark et al. "Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge". In: *arXiv e-prints* (2018).

[98] Rowan Zellers et al. "HellaSwag: Can a Machine Really Finish Your Sentence?" In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2019, pp. 4791–4800.

[99] Sébastien Bubeck et al. "Sparks of Artificial General Intelligence: Early experiments with GPT-4". In: *arXiv e-prints* (2023).

[100] Jiaan Wang et al. "Is ChatGPT a Good NLG Evaluator? A Preliminary Study". In: *arXiv e-prints* (2023).

[101] Jinlan Fu et al. "GPTScore: Evaluate as You Desire". In: *arXiv e-prints* (2023), arXiv–2302.

[102] Yen-Ting Lin and Yun-Nung Chen. "LLM-Eval: Unified Multi-Dimensional Automatic Evaluation for Open-Domain Conversations with Large Language Models". In: *Proceedings of the 5th Workshop on NLP for Conversational AI (NLP4ConvAI 2023)*. 2023, pp. 47–58.

[103] Yidong Wang et al. "PandaLM: An Automatic Evaluation Benchmark for LLM Instruction Tuning Optimization". In: *arXiv e-prints* (2023), arXiv–2306.

[104] Can Xu et al. "WizardLM: Empowering Large Language Models to Follow Complex Instructions". In: *arXiv e-prints* (2023), arXiv–2304.

[105] Anand Balachandran Pillai. *Software architecture with Python*. Packt Publishing Ltd, 2017.

[106] Mariano Anaya. *Clean Code in Python: Refactor your legacy code base*. Packt Publishing Ltd, 2018.

[107] AS Saabith, MMM Fareez, and T Vinothraj. "Python current trend applications-an overview". In: *International Journal of Advance Engineering and Research Development* 6.10 (2019).

[108] Gede Artha Azriadi Prana et al. "Categorizing the content of github readme files". In: *Empirical Software Engineering* 24 (2019). Publisher: Springer, pp. 1296–1327.

[109] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Routledge, 2013.

[110] Joshua Ainslie et al. "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 2023, pp. 4895–4901.

[111] Rewon Child et al. "Generating Long Sequences with Sparse Transformers". In: *arXiv e-prints* (2019), arXiv–1904.

[112] Niels J Blunch. "Position bias in multiple-choice questions". In: *Journal of Marketing Research* 21.2 (1984). Publisher: SAGE Publications Sage CA: Los Angeles, CA, pp. 216–220.

[113] Tri Dao. "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning". In: *arXiv e-prints* (2023), arXiv–2307.

[114] Ari Holtzman et al. "The Curious Case of Neural Text Degeneration". In: *International Conference on Learning Representations*. 2019.

A. Screenshots of Evaluation App

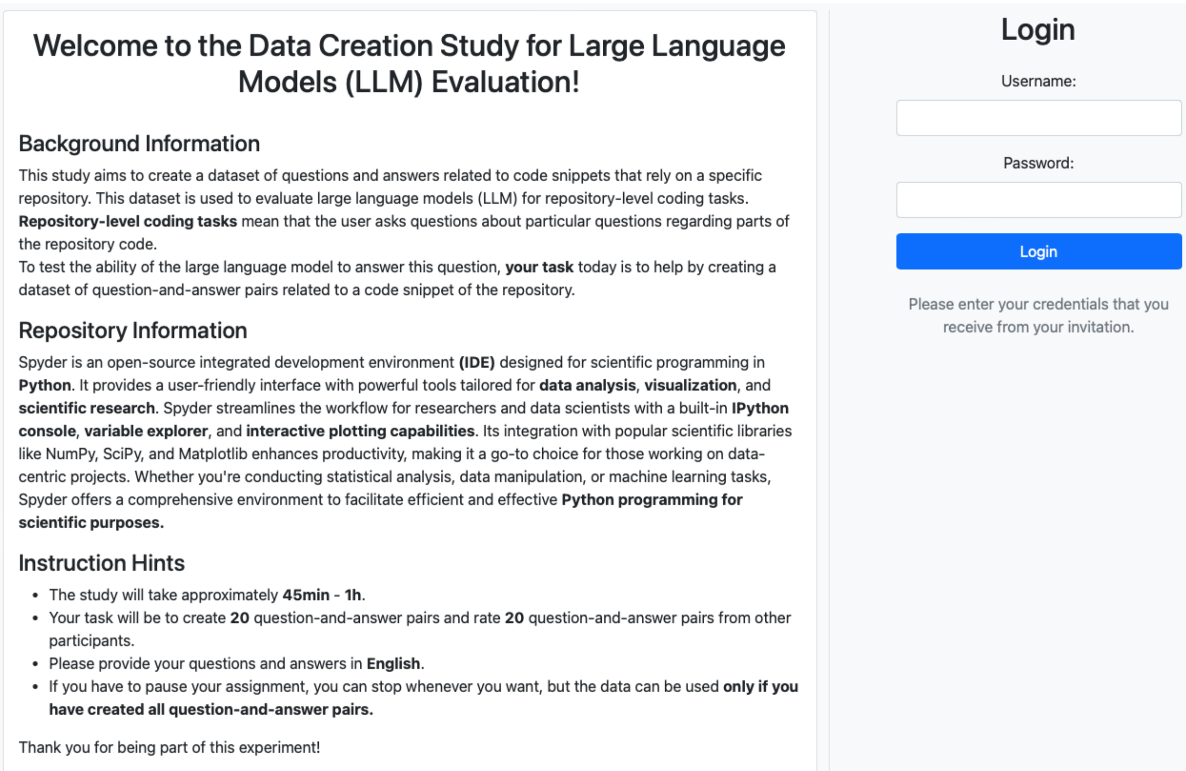

Figure A.1: Login screen for the evaluation web application

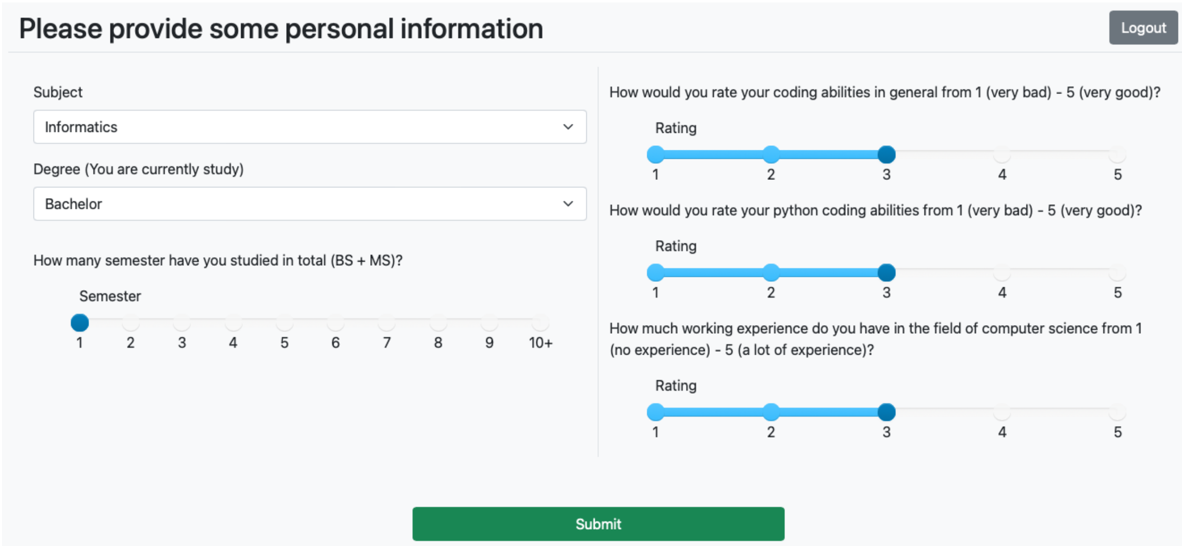

Figure A.2: User Information Screen

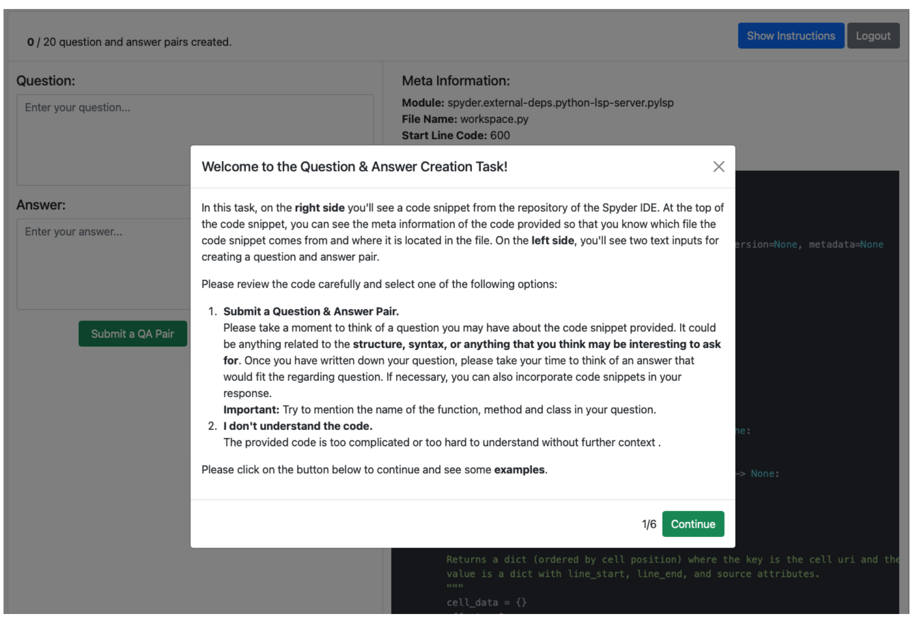

Figure A.3: Create Question-and-Answer Pairs Instruction Screen 1

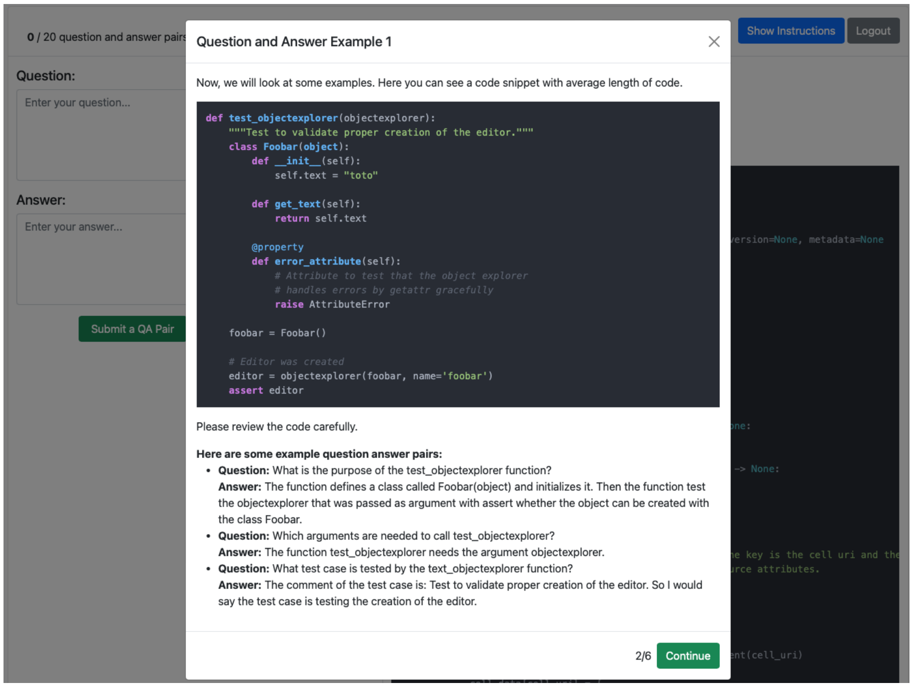

Figure A.4: Create Question-and-Answer Pairs Instruction Screen 2

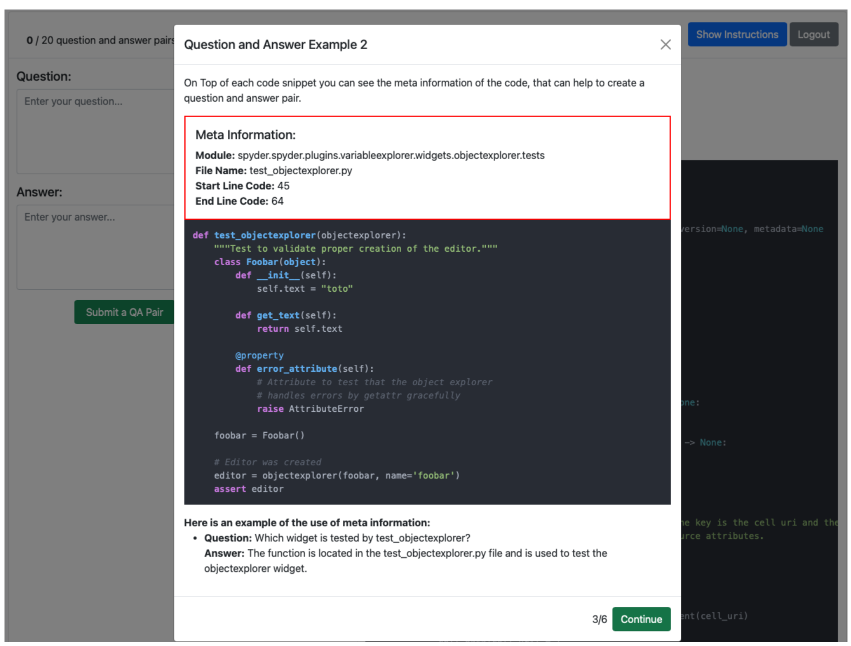

Figure A.5: Create Question-and-Answer Pairs Instruction Screen 3

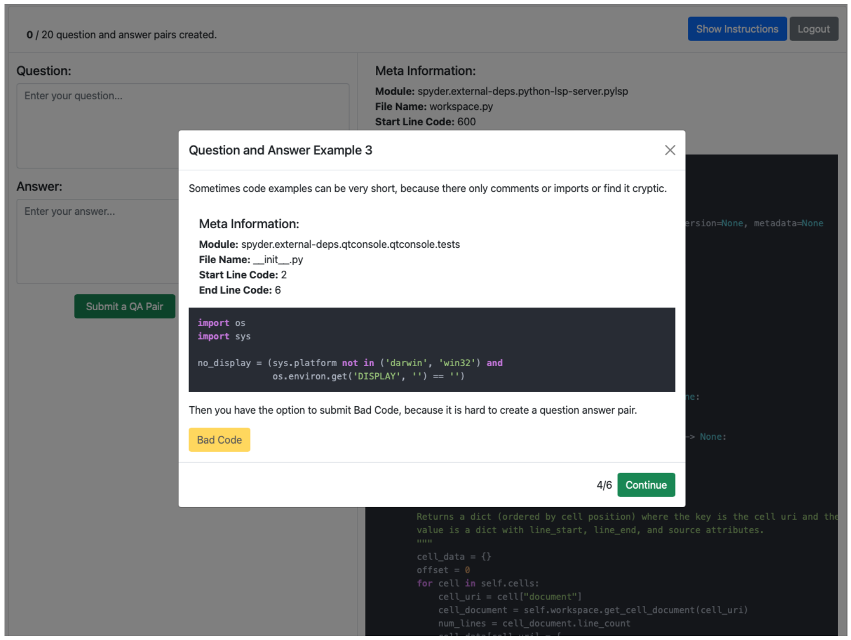

Figure A.6: Create Question-and-Answer Pairs Instruction Screen 4

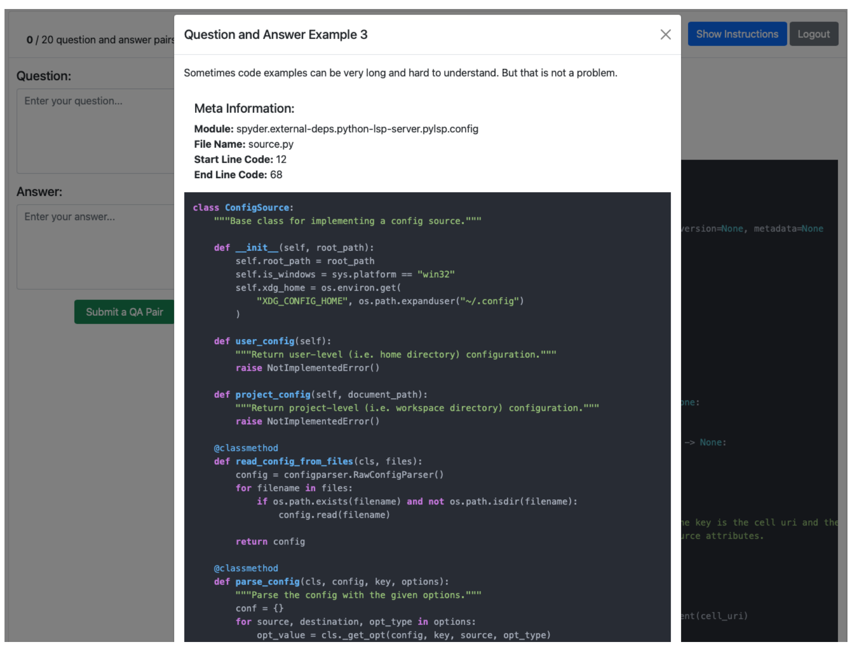

Figure A.7: Create Question-and-Answer Pairs Instruction Screen 5 1/2

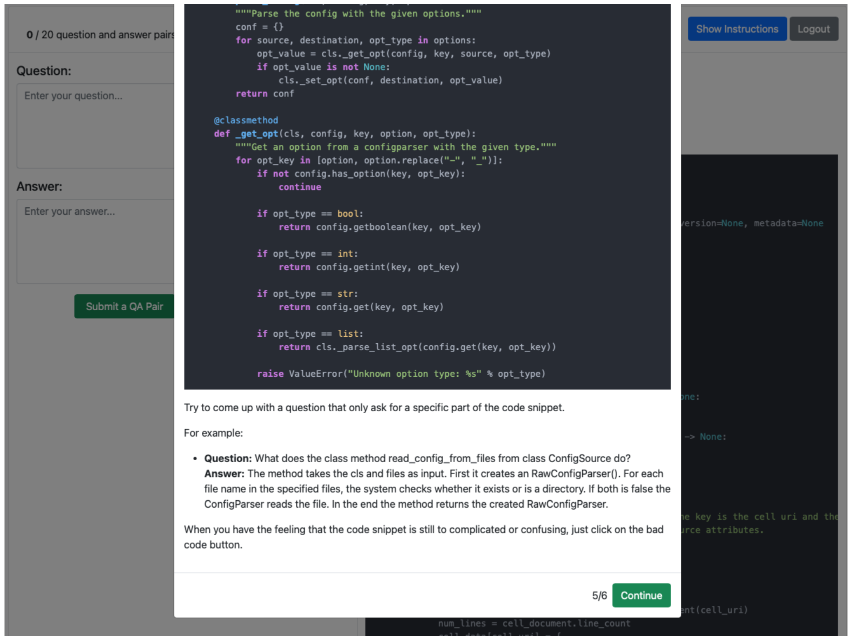

Figure A.8: Create Question-and-Answer Pairs Instruction Screen 5 2/2

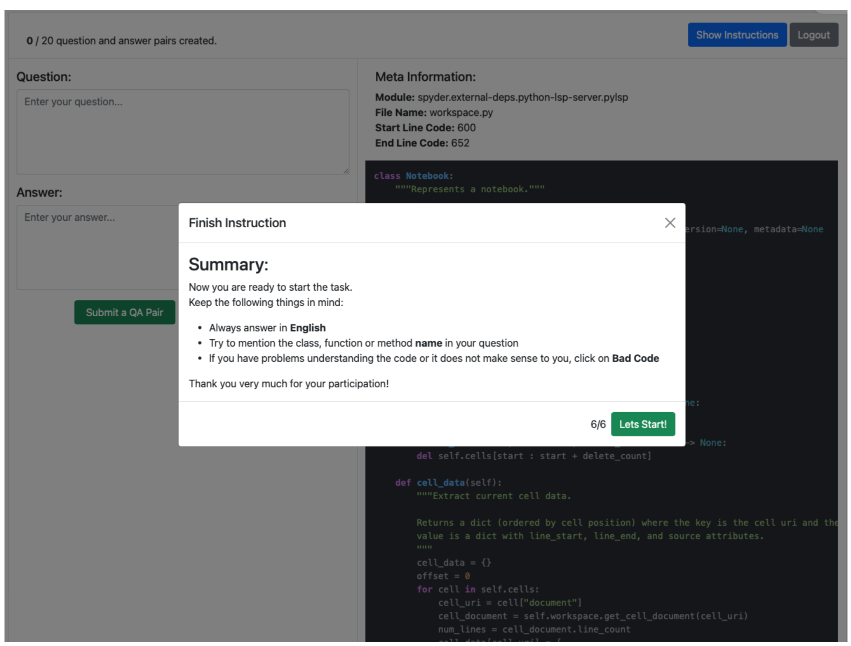

Figure A.9: Create Question-and-Answer Pairs Instruction Screen 6

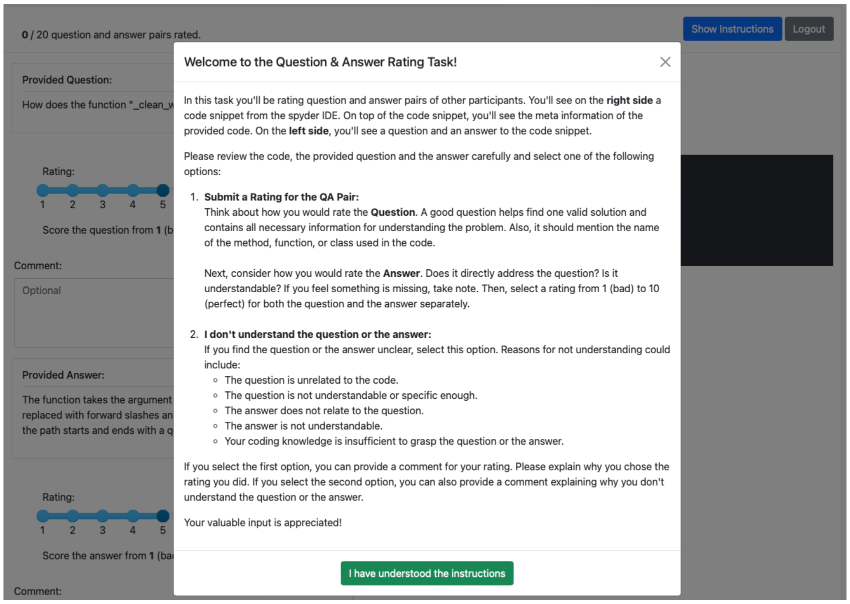

Figure A.10: Rating Question-and-Answer Pairs Instruction Screen

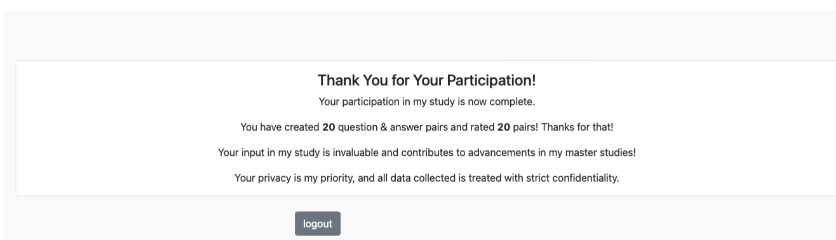

Figure A.11: End screen for the evaluation web application

B. Repository Chunks Example

```
{
    "meta_data": {
        "file_name": "requirements.py",
        "module": "spyder.spyder",
        "contains_class": false,
        "contains_function": true,
        "file_imports": [
            "from packaging.version import parse",
            "import tkinter as tk",
            "import qtpy",
        ],
        "start_line": 1,
        "end_line": 29,
    },
    "code": "
from packaging.version import parse
def show_warning(message):
    '''Show warning using Tkinter if available'''
    try:
        # If tkinter is installed (highly probable), show an error pop-up.
        # From https://stackoverflow.com/a/17280890/438386
        import tkinter as tk
        root = tk.Tk()
        root.title('Spyder')
        label = tk.Label(root, text=message, justify='left')
        label.pack(side='top', fill='both', expand=True, padx=20, pady=20)
        button = tk.Button(root, text='OK', command=root.destroy)
        button.pack(side='bottom', fill='none', expand=True)
        root.mainloop()
    except Exception:
        pass
    raise RuntimeError(message)",
}
```

Figure B.1: Example *.py* Repository chunk from the Spyder IDE. Extracted from the requirements.py, including all meta information to the file.

```
{
    "meta_data": {
        "file_name": "ISSUE_TEMPLATE.md",
        "module": "spyder..github",
        "contains_class": false,
        "contains_function": false,
        "file_imports": [],
        "start_line": 1,
        "end_line": 48,
    },
    "code": "
### Issue Report Checklist

* [ ] Searched the [issues page]
(https://github.com/spyder-ide/spyder/issues?q=is%3Aissue) for similar reports
* [ ] Read the relevant sections of the [Spyder Troubleshooting Guide]
(https://github.com/spyder-ide/spyder/wiki/Troubleshooting-Guide-and-FAQ)
and followed its advice
* [ ] Reproduced the issue after updating with ``conda update spyder``
(or ``pip``, if not using Anaconda)
* [ ] Could not reproduce inside ``jupyter qtconsole`` (if console-related)
* [ ] Tried basic troubleshooting (if a bug/error)
    * [ ] Restarted Spyder
    * [ ] Reset preferences with ``spyder --reset``
    * [ ] Reinstalled the latest version of [Anaconda]
    (https://www.anaconda.com/download/)
    * [ ] Tried the other applicable steps from the Troubleshooting Guide
* [ ] Completed the **Problem Description**, **Steps to Reproduce**
and **Version** sections below

## Problem Description

### What steps reproduce the problem?

### What is the expected output? What do you see instead?

### Paste Traceback/Error Below (if applicable)
<!--- Copy from error dialog or View > Panes > Internal Console --->
```python-traceback
PASTE TRACEBACK HERE

## Versions
<!--- You can get this information from Help > About Spyder...
or (if Spyder won't launch) the 'conda list' command
from the Anaconda Prompt/Terminal/command line. --->
[...]
"
```

Figure B.2: Example *.md* Repository chunk from the Spyder IDE. Extracted from the ISSUS_TEMPLATE.md, with all the information. Cut the last 10 lines due to space
```

C. Question Corpus

What is the name of the function/ class?
Which parameter does the function/ class has?
Which return type does the function/ class has?
Is it a Function or Class or Method?
Give me the code for the function <<name>>?
What functionality does this code aim to achieve?
What are the expected outputs or outcomes of running this code?
What variables are used in this code, and how are they defined?
What data structures are utilized, and why were they chosen?
How does the code control the flow of execution?
Are there conditional statements or loops, and how do they operate?
How does the code handle errors or unexpected situations?
Are there mechanisms in place to catch exceptions or problematic scenarios?
How might you improve the efficiency or performance of this code?
Is this code scalable for larger datasets or more complex scenarios?
How easy would it be to maintain or extend this code in the future?
Is the code adequately documented with comments or docstrings?
Are there areas where additional documentation would be beneficial?
Does this code adhere to best practices and coding standards?
Are there any deviations from commonly accepted conventions?
How are variables initialized and assigned values in the code?
Are there any variable naming conventions followed in the code?
How are comments utilized within the code?
Are there any comments explaining specific lines or blocks of code?
What are the data types used for the variables, and how are they declared?

Figure C.1: Question Corpus for Source Code Semantic

Does the code depend on external libraries or modules?
How are external dependencies managed or imported?
What external libraries or modules does the code snippet depend on?
How are the external dependencies imported within the code?
Are there any optional dependencies that are conditionally imported based on certain conditions?
How are version conflicts or compatibility issues managed with the dependencies?
Are there any considerations regarding licensing or usage restrictions for the external dependencies?

Figure C.2: Question Corpus for Dependencies

Does this code rely on specific versions of external libraries or modules?
What is the filename and module name associated with the code snippet?
Does the file contain any classes or functions?
How many lines does the code snippet span from start to end?
Is there any additional metadata or information provided about the code snippet that could be relevant for understanding its context?
How does the code snippet fit within the broader context of the module or project it belongs to?
Has the code snippet been tested, and if so, what testing methodologies were employed?

Figure C.3: Question Corpus for Meta Information

D. Prompt Templates

```
<<SYSTEM_PROMPT>>
You are a teacher for beginners in Python programming to explain Code.
First, explain from which file and module this code snippet is taken and which imports
are needed. Then, explain the code line by line.
Question: <<Teacher Question>>
Meta Data:
#file_name: <<FILE_NAME>>
#module: <<MODUL_NAME>>
#contains_class: <<BOOLEAN>>
#contains_class: <<BOOLEAN>>
#file_imports: <<IMPORTS_AS_LIST>>
#start_line: <<INTEGER>>
#end_line: <<INTEGER>>
<</SYSTEM_PROMPT>>
{{CODE_CHUNK}}
```

Figure D.1: The following is a description of the Prompt Template utilized to generate the Teacher Data D_0. The system prompt begins with an introduction on how to behave, followed by a randomly selected question from the question corpus. Additionally, the meta data for the related code chunk is included. Following the system prompt, the code chunk is added as input.

You are a model to generate a question-answer pair. You will receive an explanation of a code snippet. The provided function is Python code and is part of the Spyder IDE repository. Predict a question a user would ask. Always include the name of the file, the module in the question and the start and end line of the file. Always include in your answer code from the explanation. Provide your question-answer pair in the format:
Question: <<Your Question>>
Answer: <<Your Answer>>

Figure D.2: Prompt Template used to generate the Q&A Data D_1

Below is an instruction from an user and a candidate answer. Evaluate whether or not the answer is a good example of how AI Assistant should respond to the user's instruction. Please assign a score using the following 5-point scale: 1: It means the answer is incomplete, vague, off-topic, controversial, or not exactly what the user asked for. For example, some content seems missing, numbered list does not start from the beginning, the opening sentence repeats user's question. Or the response is from another person's perspective with their personal experience (e.g. taken from blog posts), or looks like an answer from a forum. Or it contains promotional text, navigation text, or other irrelevant information.
2: It means the answer addresses most of the asks from the user. It does not directly address the user's question. For example, it only provides a high-level methodology instead of the exact solution to user's question.
3: It means the answer is helpful but not written by an AI Assistant. It addresses all the basic asks from the user. It is complete and self contained with the drawback that the response is not written from an AI assistant's perspective, but from other people's perspective. The content looks like an excerpt from a blog post, web page, or web search results. For example, it contains personal experience or opinion, mentions comments section, or share on social media, etc.
4: It means the answer is written from an AI assistant's perspective with a clear focus of addressing the instruction. It provide a complete, clear, and comprehensive response to user's question or instruction without missing or irrelevant information. It is well organized, self-contained, and written in a helpful tone. It has minor room for improvement, e.g. more concise and focused.
5: It means it is a perfect answer from an AI Assistant. It has a clear focus on being a helpful AI Assistant, where the response looks like intentionally written to address the user's question or instruction without any irrelevant sentences. The answer provides high quality content, demonstrating expert knowledge in the area, is very well written, logical, easy-to-follow, engaging and insightful. Please first provide a brief reasoning you used to derive the rating score, and then write 'Score: <rating>' in the last line.
{Generated Q&A}

Figure D.3: Prompt Template to generating the final training dataset D_2. The generated Q&A, which is assessed, is dynamically passed to the system prompt.

Answer the question using the provided context.
Context: <<Documents>>
Question: <<Question>>

Figure D.4: Prompt Template to generate the response after retrieving the chunk from the vector database. <<Documents>> are the retrieved documents. <<Question>> is the question by the user's request.

<<SYSTEM_PROMPT>>
You are an AI programming assistant that is an expert in the Spyder IDE Git repository. Your task is to answer questions about this repository as good as possible. Consider the following information about the repository. The repository is open-source and hosted on GitHub. Anybody can contribute to the codebase.
Please only give truthful answers, and if you don't know an answer, don't hallucinate, but write that you don't know it.
<< /SYSTEM_PROMPT>>
[User Question] <<USER_QUESTION>> [End of User Question]
[/INST]

Figure D.5: Overview of the prompt template used to generate the responses for the LLM-as-a-Judge evaluation. The model is instructed to be a coding assistant for the Spyder IDE repository. The task is to answer questions about the repository. Also, the model is reminded to always tell the truth and not hallucinate.

<<SYSTEM PROMPT>>
Please act as an impartial judge and evaluate the quality of the responses provided by two AI assistants to the user question and the model solution displayed below. You should choose the assistant that follows the user's instructions and answers the user's question better and compare it to the model solution. Your evaluation should consider factors such as the helpfulness, relevance, accuracy, depth, creativity, and level of detail of their responses. Begin your evaluation by comparing the two responses and provide a short explanation. Think step by step. Avoid any position biases and ensure that the order in which the responses were presented does not influence your decision. Do not allow the length of the responses to influence your evaluation. Do not favor certain names of the assistants. Be as objective as possible. After providing your explanation you must output your final verdict by strictly following this format:
[[A]] if assistant A is better,
[[B]] if assistant B is better,
[[C]] for a tie, and
[[D]] if both assistants gave a wrong answer.
<</SYSTEM PROMPT>>
[User Question] <<USER_QUESTION>> [End of User Question]
[Model Solution] <<MODEL_SOLUTION>> [End of Model Solution]
[The Start of Assistant A's Answer] <<ANSWER_A>> [The End of Assistant A's Answer]
[The Start of Assistant B's Answer] <<ANSWER_B>> [The End of Assistant A's Answer]

Figure D.6: Overview of the prompt template used to execute the model-based pairwise comparison evaluation. First, the system prompt is shown. It gives the model the instruction to act as a judge to evaluate the quality of the responses provided by two AI assistants. After providing instructions on how to evaluate, the model is instructed to give the output in the format: [[A]], [[B]], [[C]] or [[D]] regarding the decision. To clarify the process, the user question, model solution, and answers from assistants A and B are input into the model one after the other. Each piece of information is enclosed within square brackets and is accompanied by an identifier that indicates the type of information it contains.

E. Training Conditions

E.1 Loss Curves

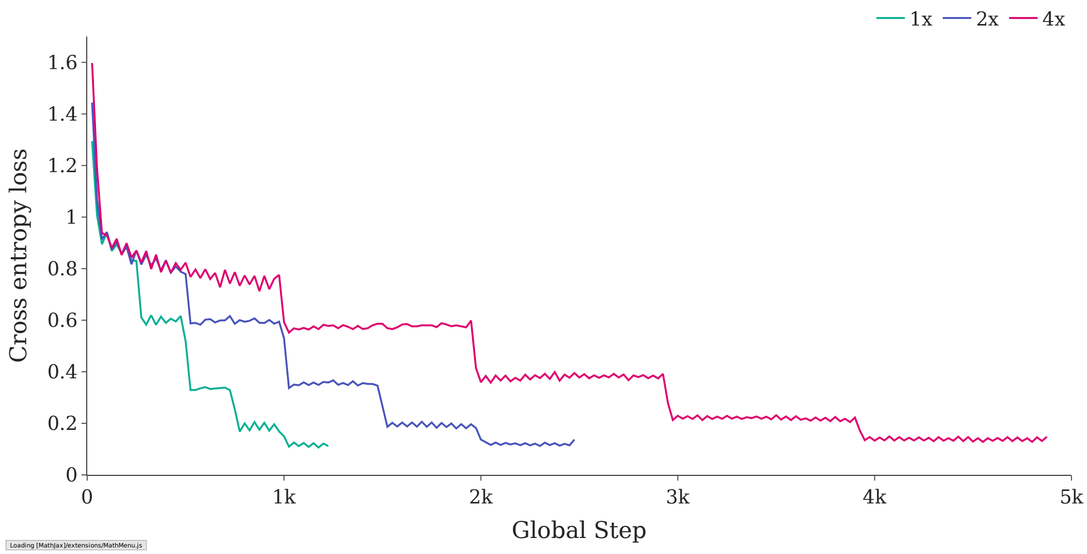

Figure E.1: During the training process of 5 epochs, the cross entropy loss development value is demonstrated. Each line represents one training run. "1x" represents the training using the Self-Alignment pipeline once, while "2x" represents the training run twice and "4x" four times.

E.2 Learning Rate

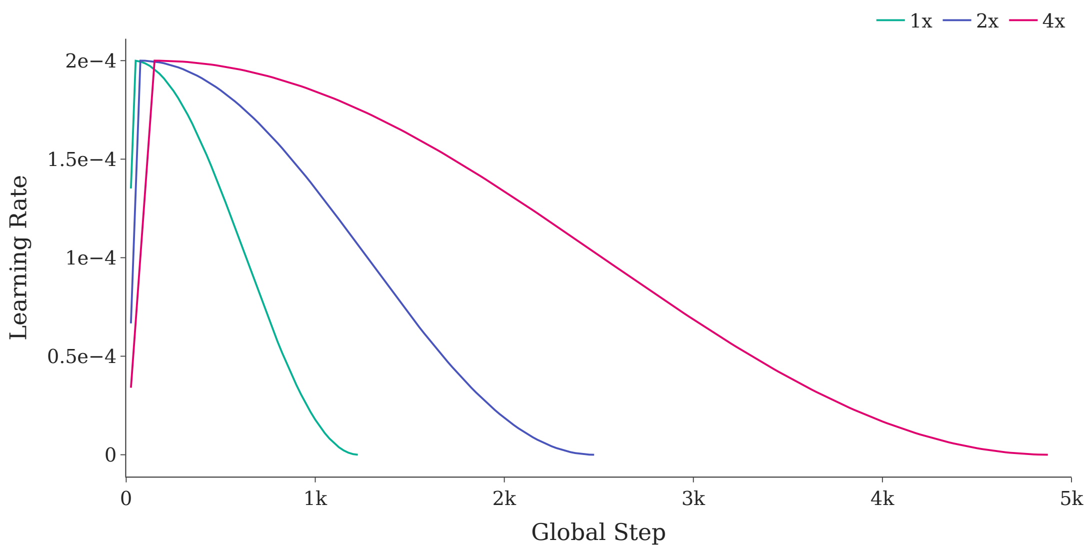

Figure E.2: During the training process of 5 epochs, the value of the learning rate development is demonstrated. Each line represents one training run. "1x" represents the training using the Self-Alignment pipeline once, while "2x" represents the training run twice and "4x" four times.

F. Q&A Pairs from the LLM-as-a-Judge evaluation

Q&A Pair 135 (Code Semantics): Fine-Tuned Model

Question:
What does the "InstallerMissingDependencies" class do?

Correct Answer:
The "InstallerMissingDependencies" class holds a function "_msg" that takes in an error message that is likely to contain a list of missing dependencies for the installation of Spyder. The function then formats the error message and provides additional context to the message before returning it.

Mistral 7B (A):
I cannot definitively answer that question without access to the specific codebase in the Spyder IDE Git repository that contains the "InstallerMissingDependencies" class. However, based on the name, it appears that this class may be related to handling missing dependencies during the installation process in Spyder. If someone could provide the context or location of this class within the codebase, I would be happy to try and provide a more accurate answer. If you don't have that information,
I'm afraid I cannot provide a definitive answer.

Fine-Tuned Model (B):
The "InstallerMissingDependencies" class is a custom exception class defined in the Spyder IDE project. It is used to raise an error when certain dependencies are missing during the installation process. Here is the code snippet for this class::

```
class InstallerMissingDependencies(Exception):
    def __init__(self, missing_dependencies, installer_type):
        super().__init__("Missing dependencies: {0}".format(missing_dependencies))
        self.missing_dependencies = missing_dependencies
        self.installer_type = installer_type

    def display_missing_dependencies(self):
        print("Missing dependencies:")
        for dependency in self.missing_dependencies:
            print("* {0}".format(dependency))
```

Judgement:
Assistant A provides a thoughtful response by stating that they cannot definitively answer the question without access to the specific codebase, but they make an educated guess based on the name of the class. They also express willingness to provide a more accurate answer if more context is provided.
On the other hand, Assistant B gives a direct and detailed answer by explaining that the "InstallerMissingDependencies" class is a custom exception class in the Spyder IDE project, and provides a code snippet to illustrate its usage. Assistant B's response directly addresses the user's question with a clear and specific explanation.
Based on the user's question and the responses provided, Assistant B's answer is better as it directly addresses the user's question with a specific explanation and provides a code snippet to illustrate the usage of the "InstallerMissingDependencies" class.
Therefore, the final verdict is: "**[[B]]**" Assistant B is better.

Figure F.1: Q&A Pair 135 from the SpyderCodeQA answered by Mistral 7B and Fine-Tuned Model.

Q&A Pair 135 (Code Semantics): RAG Pipeline

Question:
What does the "InstallerMissingDependencies" class do?

Correct Answer:
The "InstallerMissingDependencies" class holds a function "_msg" that takes in an error message that is likely to contain a list of missing dependencies for the installation of Spyder. The function then formats the error message and provides additional context to the message before returning it.

Mistral 7B (A):
I cannot definitively answer that question without access to the specific codebase in the Spyder IDE Git repository that contains the "InstallerMissingDependencies" class. However, based on the name, it appears that this class may be related to handling missing dependencies during the installation process in Spyder. If someone could provide the context or location of this class within the codebase, I would be happy to try and provide a more accurate answer. If you don't have that information, I'm afraid I cannot provide a definitive answer..

Context from RAG:

```
import os
import glob
import textwrap
import logging

from spyder.config.base import get_conf_path


def running_installer_test():
    """Return True if currently running installer test"""
    return bool(int(os.environ.get('INSTALLER_TEST', '0')))
...
...
...
class InstallerMissingDependencies(SpyderInstallerError):
    """Error for missing dependencies"""
    def _msg(self, msg):
        msg = msg.replace('<br>', '\n')
        msg = 'Missing dependencies' + textwrap.
          indent(msg, '  ')

        return msg
```

Mistral 7B + RAG (B):
The "InstallerMissingDependencies" class is a subclass of "SpyderInstallerError" and is used to raise an error when some dependencies are missing during the installation process of Spyder. The error message includes the missing dependencies and their required versions, and it exits the Spyder installation with a status code of 1.

Judgement:
Assistant A provides a cautious response, stating that they cannot definitively answer the question without access to the specific codebase in the Spyder IDE Git repository that contains the "InstallerMissingDependencies" class. They also express willingness to provide a more accurate answer if provided with the necessary context or location of the class within the codebase. Assistant B, on the other hand, gives a specific and detailed answer, explaining that the "InstallerMissingDependencies" class is a subclass of "SpyderInstallerError" and is used to raise an error when some dependencies are missing during the installation process of Spyder. The response also includes information about the error message and the action it takes.
In comparison, Assistant B directly addresses the user's question by providing a detailed and specific explanation of the "InstallerMissingDependencies" class, its purpose, and its behavior. Assistant A, while cautious, does not provide as much specific information about the class.

Therefore, Assistant B provides a better answer to the user's question.
Therefore, the final verdict is: "**[[B]]**" Assistant B is better.

Figure F.2: Q&A Pair 135 from the SpyderCodeQA answered by Mistral 7B and Mistral 7B with RAG pipeline.

Q&A Pair 135 (Code Semantics): Fine-Tuned + RAG pipeline

Question:
What does the "InstallerMissingDependencies" class do?

Correct Answer:
The "InstallerMissingDependencies" class holds a function "_msg" that takes in an error message that is likely to contain a list of missing dependencies for the installation of Spyder. The function then formats the error message and provides additional context to the message before returning it.

Mistral 7B (A):
I cannot definitively answer that question without access to the specific codebase in the Spyder IDE Git repository that contains the "InstallerMissingDependencies" class. However, based on the name, it appearsthat this class may be related to handling missing dependencies during the installation process in Spyder. If someone could provide the context or location of this class within the codebase, I would be happy to try and provide a more accurate answer.
If you don't have that information, I'm afraid I cannot provide a definitive answer.

Context from RAG:

```
import os
import glob
import textwrap
import logging

from spyder.config.base import get_conf_path


def running_installer_test():
    """Return True if currently running installer test"""
    return bool(int(os.environ.get('INSTALLER_TEST', '0')))
...
...
...
class InstallerMissingDependencies(SpyderInstallerError):
    """Error for missing dependencies"""
    def _msg(self, msg):
        msg = msg.replace('<br>', '\n')
        msg = 'Missing dependencies' + textwrap.
          indent(msg, '  ')

        return msg
```

Fine-Tuned Model + RAG (B):
The "InstallerMissingDependencies" class is an error class that inherits from "SpyderInstallerError". It is used to represent errors related to missing dependencies. The class defines a constructor that takes a message as an argument and writes the error message to the console. The error message is formatted with a newline charactr instead of a '<br>' tag for better display. Here's the code snippet:

```
class InstallerMissingDependencies(SpyderInstallerError):
    """Error for missing dependencies"""

   def _msg(self, msg):
     msg = msg.replace('<br>', '\n')
     msg = 'Missing dependencies' + textwrap.indent(msg, '  ')

     return msg
```

Judgement:
Assistant A's response is cautious and lacks specific details about the "InstallerMissingDependencies" class.
It mentions the potential relation to handling missing dependencies during installation in Spyder but fails to provide concrete information about the class's functionality. The assistant also mentions its inability to access the specific codebase, which, while honest, does not help answer the user's question effectively.
Assistant B, on the other hand, provides a detailed and specific answer about the "InstallerMissingDependencies" class.
It includes information about the class being an error class, its inheritance from "SpyderInstallerError," and its purpose to represent errors related to missing dependencies. Furthermore, Assistant B offers a code snippet that shows how the class formats an error message, which aligns well with the model solution's description of the class handling error messages about missing dependencies. This response is not only relevant and accurate but also provides depth by including a code example. Comparing both responses to the model solution, Assistant B's response is more aligned with the model solution in terms of accuracy, relevance, and depth. Assistant A's response, while cautious, does not provide the necessary details to be helpful or accurate in the context of the user's question. Final Verdict: **[[B]]**

Figure F.3: Q&A Pair 135 from the SpyderCodeQA answered by Mistral 7B and Fine-Tuned Model with RAG pipeline.

Q&A Pair 2 (Code Semantics): Fine-Tuned + RAG pipeline

Question:

The test_cython_client function is part of the test_ipythonconsole.py file.
What is the main purpose of the test function?

Correct Answer:

The main purpose of the test function is to test wherever the Cython console is working correctly.

Mistral 7B (A):

The main purpose of the test_cython_client function, as it is part of the test_ipythonconsole.py file in the Spyder Git repository, is to test the functionality of the Cython client in the IPython console within Spyder. Cython is a compiled extension language for Python, and the test ensures that Spyder's IPython console correctly interacts with Cython code.

Context from RAG:

```
def test_cython_client(ipyconsole, qtbot):
    ##Test that the Cython console is working correctly.
    # Wait until the window is fully up
    shell = ipyconsole.get_current_shellwidget()

    # This is here to generate further errors
    with qtbot.waitSignal(shell.executed,
                               timeout=SHELL_TIMEOUT):
        shell.execute("%%cython\n"
                      "cdef int ctest(int x, int y):\n"
                      "    return x + y")

    # Assert there are no errors in the console
    control = ipyconsole.get_widget().get_focus_widget()
    assert 'Error' not in control.toPlainText()

    # Reset the console namespace
    shell.reset_namespace()
    qtbot.wait(1000)

    # See that cython is still enabled after reset
    with qtbot.waitSignal(shell.executed,
                              timeout=SHELL_TIMEOUT):
        shell.execute("%%cython\n"
                      "cdef int ctest(int x, int y):\n"
                      "    return x + y")

    # Assert there are no errors after restting the console
    control = ipyconsole.get_widget().get_focus_widget()
    assert 'Error' not in control.toPlainText()
```

Fine-Tuned Model + RAG (B):

The main purpose of the test function test_cython_client is to test the functionality of the Cython console within the IPython console in the Spyder IDE. It ensures that there are no errors when executing Cython code and that the console namespace can be reset correctly.

Here is the code snippet for reference:

```
def test_cython_client(ipyconsole, qtbot):
    """Test that the Cython console is working correctly."""
    # Wait until the window is fully up
    shell = ipyconsole.get_current_shellwidget()
...
...
```

Judgement:

Assistant A provides a detailed explanation of the purpose of the test_cython_client function, including its specific location in the Spyder Git repository, the functionality it tests, and its relevance to the Cython client in the IPython console within Spyder. The assistant also includes a code snippet for reference.
Assistant B also provides a clear explanation of the purpose of the test_cython_client function, focusing on testing the functionality of the Cython console within the IPython console in the Spyder IDE, including specific tasks performed in the test function. The assistant also includes a code snippet for reference.
Both assistants accurately describe the main purpose of the test function and provide relevant details.
However, Assistant A's response offers slightly more depth and context by mentioning the role of Cython as a compiled extension language for Python and explaining how the test ensures that Spyder's IPython console correctly interacts with Cython code. Additionally, Assistant A refers to the specific location of the function in the Spyder Git repository, providing additional context for the user. Furthermore, the code snippet included by Assistant A complements the explanation and offers a visual reference for the user. Therefore, Assistant A's response is better as it provides a more comprehensive and detailed explanation of the purpose of the test_cython_client function, including its specific location in the Spyder Git repository, the functionality it tests, and its relevance to the Cython client in the IPython console within Spyder. Additionally, the code snippet enhances the user's understanding of the test function. **[[A]]**

Figure F.4: Q&A Pair 2 from the SpyderCodeQA answered by Mistral 7B and Fine-Tuned Model with RAG pipeline.

Q&A Pair 2 (Code Semantics): Fine-Tuned + RAG pipeline (Judged by GPT-4)

Question:
The test_cython_client function is part of the test_ipythonconsole.py file.
What is the main purpose of the test function?

Correct Answer:
The main purpose of the test function is to test wherever the Cython console is working correctly.

Mistral 7B (A):
The main purpose of the test_cython_client function, as it is part of the test_ipythonconsole.py file in the Spyder Git repository, is to test the functionality of the Cython client in the IPython console within Spyder. Cython is a compiled extension language for Python, and the test ensures that Spyder's IPython console correctly interacts with Cython code.

Context from RAG:

```
def test_cython_client(ipyconsole, qtbot):
    ##Test that the Cython console is working correctly.
    # Wait until the window is fully up
    shell = ipyconsole.get_current_shellwidget()

    # This is here to generate further errors
    with qtbot.waitSignal(shell.executed,
                          timeout=SHELL_TIMEOUT):
        shell.execute("%%cython\n"
                      "cdef int ctest(int x, int y):\n"
                      "    return x + y")

    # Assert there are no errors in the console
    control = ipyconsole.get_widget().get_focus_widget()
    assert 'Error' not in control.toPlainText()

    # Reset the console namespace
    shell.reset_namespace()
    qtbot.wait(1000)

    # See that cython is still enabled after reset
    with qtbot.waitSignal(shell.executed,
                         timeout=SHELL_TIMEOUT):
        shell.execute("%%cython\n"
                      "cdef int ctest(int x, int y):\n"
                      "    return x + y")

    # Assert there are no errors after restting the console
    control = ipyconsole.get_widget().get_focus_widget()
    assert 'Error' not in control.toPlainText()
```

Fine-Tuned Model + RAG (B):
The main purpose of the test function test_cython_client is to test the functionality of the Cython console within the IPython console in the Spyder IDE. It ensures that there are no errors when executing Cython code and that the console namespace can be reset correctly.

Here is the code snippet for reference:

```
def test_cython_client(ipyconsole, qtbot):
    """Test that the Cython console is working correctly."""
    # Wait until the window is fully up
    shell = ipyconsole.get_current_shellwidget()
...
...
```

Judgement:
Relevance and Accuracy: Both assistants correctly identify the main purpose of the test_cython_client function. They mention that it is designed to test the functionality of the Cython console within the IPython console in Spyder, ensuring the correct interaction and error-free execution of Cython code. Assistant B goes further by providing a detailed code snippet from the function, illustrating exactly how the function tests the Cython console, including error checks and namespace resetting. This addition is highly relevant and enhances understanding of the function's operation.
Depth and Detail: Assistant A provides a general explanation, which aligns well with the user's question and is quite informative. Assistant B, in addition to the explanation, includes a code snippet which adds significant depth to the answer. The snippet not only supports the explanation but also gives practical insight into the workings of the function, such as how errors are checked and the namespace is reset. **Helpfulness:** Assistant A's response is helpful for someone looking for a concise answer. Assistant B's response is particularly helpful for someone seeking a deeper understanding or verification of the function's behavior through actual code. **Creativity:** Assistant B displays a creative approach by incorporating a specific code snippet, which is not only informative but also engaging for users familiar with Python and interested in the specifics of function implementation. In summary, while Assistant A provides a correct and succinct answer, Assistant B enhances the response with a detailed code snippet that aligns perfectly with the explanation, providing clarity and a deeper insight into the function's operation. This makes Assistant B's answer more comprehensive and informative. Final Verdict: **[[B]]**

Figure F.5: Q&A Pair 2 from the SpyderCodeQA answered by Mistral 7B and Fine-Tuned Model with RAG pipeline. Judged by GPT-4.

Q&A Pair 211 (Dependencies): Fine-Tuned Model

Question:
What library import is Used in the file test_introspection.py?

Correct Answer:
The libraries used in the file test_introspection.py are os, os.path, random, textwrap, sys, flaky, pytest, pytestqt, Qt, and QTextCursor.

Mistral 7B (A):
To answer your question, I would need to have access to the specific test_introspection.py file in the Spyder IDE Git repository. Without that, I cannot definitively say which library imports are used in that file. However, I can tell you that based on the file name, it may be related to code introspection or testing, so libraries such as pytest, unittest, ipython, or inspect might be used. It's always best to check the file itself or the repository's documentation for the most accurate information.

Fine-Tuned Model (B):
The file test_introspection.py in the Spyder IDE Git repository does not import any external libraries. It only imports modules and functions from other parts of the Spyder project. Here is the list of imports at the beginning of the file:

```
from spyder.plugins.editor.widgets.codeeditor import CodeEditor
from spyder.plugins.completion.api import CompletionRequestTypes
from spyder.plugins.completion.providers.languageserver.providers.utils import path_as_uri
from spyder.utils.conda import get_list_conda_envs
```

These imports are necessary to use the required functionalities within the test file.

Judgement:
Assistant A provides a comprehensive and thoughtful response. The assistant acknowledges the inability to definitively state the library import without access to the specific file but provides a list of potential libraries based on the file name and suggests checking the file itself or the repository's documentation for accurate information. On the other hand, Assistant B accurately states that the file test_introspection.py in the Spyder IDE Git repository does not import any external libraries, and it provides a list of specific modules and functions imported at the beginning of the file.
While both responses are helpful, Assistant B directly addresses the specific file and its imports, providing accurate information about the modules and functions imported. Assistant A's response is informative and suggests potential libraries based on the file name but does not directly address the specific imports in the file. Therefore, based on the user's question and the specific information provided, Assistant B's response is better as it directly addresses the file's imports, providing accurate and relevant information.
Therefore, the final verdict is: **[[B]]**

Figure F.6: Q&A Pair 211 from the SpyderCodeQA answered by Mistral 7B and Fine-Tuned Model.

Q&A Pair 211 (Dependencies): RAG Pipeline

Question:
What library import is Used in the file test_introspection.py?

Correct Answer:
The libraries used in the file test_introspection.py are os, os.path, random, textwrap, sys, flaky, pytest, pytestqt, Qt, and QTextCursor.

Mistral 7B (A):
To answer your question, I would need to have access to the specific test_introspection.py file in the Spyder IDE Git repository. Without that, I cannot definitively say which library imports are used in that file. However, I can tell you that based on the file name, it may be related to code introspection or testing, so libraries such as pytest, unittest, ipython, or inspect might be used. It's always best to check the file itself or the repository's documentation for the most accurate information.

Context from RAG:

```
# Standard library imports
import os
import os.path as osp
import random
import textwrap
import sys

# Third party imports
from flaky import flaky
import pytest
import pytestqt
from qtpy.QtCore import Qt
from qtpy.QtGui import QTextCursor

# Local imports
from spyder.config.base import running_in_ci
from spyder.config.manager import CONF
from spyder.config.utils import is_anaconda
from spyder.plugins.completion.api import (
    CompletionRequestTypes,
CompletionItemKind)
from spyder.plugins.completion.providers.
languageserver.providers.utils import (
    path_as_uri)
from spyder.utils.conda import
get_list_conda_envs
# Location of this file
LOCATION = osp.realpath(osp.join(os.getcwd(),
osp.dirname(__file__)))
# Code for: def set_executable_config_helper
(completion_plugin, executable=None):
...
```

(Implementation Code)

Mistral 7B + RAG (B):
The file "test_introspection.py" imports the following libraries:

```
import os; import os.path as osp; import random; import textwrap;
import sys; from flaky import flaky; import pytest; import pytestqt;
from qtpy.QtCore import Qt; from qtpy.QtGui import QTextCursor;
from spyder.config.base import running_in_ci
from spyder.config.manager import CONF
from spyder.config.utils import is_anaconda
```

These imports are used throughout the file for various purposes, including file system manipulation, testing, and interacting with Spyder's configuration and plugins.

Judgement:
Assistant A's response provides a thoughtful and detailed answer to the user's question. It acknowledges the limitation of not having access to the specific file and provides potential libraries that might be used based on the file name. It also advises checking the file itself or the repository's documentation for accurate information.
Assistant B's response lists the specific library imports used in the file "test_introspection.py" and briefly explains their purposes.
Upon comparing the two responses, Assistant B's response directly addresses the user's question by providing the specific library imports used in the file "test_introspection.py," along with a brief explanation of their purposes. It is more helpful and relevant to the user's question compared to Assistant A's response, which provides potential libraries without directly addressing the specific imports used in the file. Therefore, Assistant B's response is better.
Therefore, the final verdict is: **[[B]]**

Figure F.7: Q&A Pair 211 from the SpyderCodeQA answered by Mistral 7B and Mistral 7B with RAG pipeline.

Q&A Pair 211 (Dependencies): Fine-Tuned + RAG pipeline

Question:
What library import is Used in the file ttest_introspection.py?

Correct Answer:
The libraries used in the file test_introspection.py are os, os.path, random, textwrap, sys, flaky, pytest, pytestqt, Qt, and QTextCursor.

Mistral 7B (A):
To answer your question, I would need to have access to the specific test_introspection.py file in the Spyder IDE Git repository. Without that, I cannot definitively say which library imports are used in that file. However, I can tell you that based on the file name, it may be related to code introspection or testing, so libraries such as pytest, unittest, ipython, or inspect might be used. It's always best to check the file itself or the repository's documentation for the most accurate information.

Context from RAG:

```
# Standard library imports
import os
import os.path as osp
import random
import textwrap
import sys

# Third party imports
from flaky import flaky
import pytest
import pytestqt
from qtpy.QtCore import Qt
from qtpy.QtGui import QTextCursor

# Local imports
from spyder.config.base import running_in_ci
from spyder.config.manager import CONF
from spyder.config.utils import is_anaconda
from spyder.plugins.completion.api import (
    CompletionRequestTypes,
CompletionItemKind)
from spyder.plugins.completion.providers.
languageserver.providers.utils import (
    path_as_uri)
from spyder.utils.conda import
get_list_conda_envs
# Location of this file
LOCATION = osp.realpath(osp.join(os.getcwd(),
osp.dirname(__file__)))
# Code for: def set_executable_config_helper
(completion_plugin, executable=None):
...
```

(Implementation Code)

Fine-Tuned Model + RAG (B):
The file test_introspection.py imports the following libraries:
standard library -
os, os.path, random, textwrap, sys;
third party -
flaky, pytest, pytestqt, qtpy.QtCore, qtpy.QtGui.

Judgement:
Assistant A's answer provides a thoughtful explanation of the potential libraries that could be used in the file test_introspection.py, given its name. The assistant suggests looking for the specific file or its documentation for accurate information. Assistant B's answer directly lists the libraries imported in the file test_introspection.py, providing a clear and concise response to the user's question.

Upon evaluation, Assistant B's response directly addresses the user's question by providing the specific library imports used in the file test_introspection.py, aligning with the user's request for the library import. Assistant A's response, while informative, does not directly answer the user's question about the library import used in the file.
Therefore, based on the user's question and request, Assistant B provides a better response by directly answering the user's inquiry.
Therefore, the final verdict is: **[[B]]**

Figure F.8: Q&A Pair 211 from the SpyderCodeQA answered by Mistral 7B and Fine-Tuned Model with RAG pipeline.

Q&A Pair 313 (Meta-Information): Fine-Tuned Model

Question:
What is the minimum size for the primary Spyder logo and the logomark?

Correct Answer:
The minimum size for the primary Spyder logo is 75 px wide, and for the logomark, it is 14 px wide.

Mistral 7B (A):
I cannot directly provide you with the minimum size for the Spyder IDE logo and logomark as I don't have access to the specific design files or brand guidelines of the Spyder project. However, you can refer to the official Spyder IDE GitHub repository or contact the project maintainers for any specifications related to logo usage. You can find the Spyder IDE repository on GitHub at https://github.com/spyder-ide/spyder. If you need help with using Git or contributing to the repository, feel free to ask.

Fine-Tuned Model (B):
The minimum size for the primary Spyder logo is 1500px in width, and the logomark has a minimum size of 500px in width.
Here's the relevant code snippet from the "readme.md" file:

Logos
The minimum size for the primary Spyder logo is 1500px in width.
The logomark has a minimum size of 500px in width.
Please find the logo files in the `spyder-logo` directory.

Judgement:
Assistant A provided a helpful response by acknowledging that it cannot directly provide the specific size for the Spyder IDE logo and logomark but directed the user to the official Spyder IDE GitHub repository or project maintainers for the information. Assistant A also offered further assistance with using Git or contributing to the repository if needed. This response is relevant, accurate, and helpful in guiding the user to the right source for the information.

Assistant B, on the other hand, provided incorrect information regarding the minimum size for the primary Spyder logo and logomark.
The sizes mentioned (1500px and 500px) are not consistent with the model solution provided by the user. Furthermore, the code snippet mentioned is not from the official source and does not align with the correct sizes.
Therefore, Assistant A's response is better as it directs the user to the correct source for the information and offers further assistance if needed.

Therefore, the final verdict is: **[[A]]**

Figure F.9: Q&A Pair 313 from the SpyderCodeQA answered by Mistral 7B and Fine-Tuned Model.

Q&A Pair 313 (Meta-Information): Fine-Tuned / Mistral 7B + RAG Pipeline

Question:
What is the minimum size for the primary Spyder logo and the logomark?

Correct Answer:
The minimum size for the primary Spyder logo is 75 px wide, and for the logomark, it is 14 px wide.

Mistral 7B (A):
I cannot directly provide you with the minimum size for the Spyder IDE logo and logomark as I don't have access to the specific design files or brand guidelines of the Spyder project. However, you can refer to the official Spyder IDE GitHub repository or contact the project maintainers for any specifications related to logo usage. You can find the Spyder IDE repository on GitHub at https://github.com/spyder-ide/spyder. If you need help with using Git or contributing to the repository, feel free to ask.

Context from RAG:

```
<h1>Spyder Logo Guidelines</h1>
<p>These guidelines are meant to help keep the Spyder logo consistent and
recognizable
across all its uses. They also provide a common language for referring to
the logos and their components.</p>
<p>The primary logo is the horizontal option (logomark and text next to
each other). You may also use the logomark on its own (meaning the
logomark without text).
When in doubt, it's preferable to use the primary logo that includes
text over the logomark alone. </p>
<p>If you have a case where you do not know what background the logo will
be on or it is a variable background (such as the application icon on a
desktop), it is best to use a logo option with the light grey background.
</p>
<h2>Color</h2>
<p>Colors used in the various logos are as follows:
- Full color options are a combination of red (<code>#8c0000</code> or
<code>rgb(140, 0, 0)</code>) and grey (<code>#303030</code> or
<code>rgb(48,48,48)</code>).
- Full color options with a background have the same red and grey as other
full color options and add a light grey (<code>#fafafa</code> or
<code>rgb(250, 250, 250)</code>) rectangle behind it.
- Light options are white (<code>#ffffff</code> or <code>rgb(255, 255,
255)</code>).
- Dark options are black (<code>#000000</code> or <code>rgb(0, 0, 0)
</code>).</p>
<p>Whenever possible, use the full color logos. One color logos
(light or dark) are to be used when full color will not have enough
contrast or cannot be produced.</p>
<h2>Minimum Size</h2>
<p>Please do not make the primary logo smaller than 75 px wide or the
logomark smaller than 14 px wide.</p>
<h2>Logo Integrity</h2>
<p>A few other notes to keep in mind when using the logo:
- Be sure to scale the logo proportionally.
- Maintain a good amount of space around the logo. Don't let it overlap
with text, images, or other elements without giving it a single color
background.
</p>
```

Fine-Tuned Model + RAG (B):
The minimum size for the primary Spyder logo is 75 px wide, and the minimum size for the logomark is 14 px wide.

Judgement:
Assistant A's response is cautious and lacks specific details about the "InstallerMissingDependencies" class.
It mentions the potential relation to handling missing dependencies during installation in Spyder but fails to provide concrete information about the class's functionality. The assistant also mentions its inability to access the specific codebase, which, while honest, does not help answer the user's question effectively.
Assistant B, on the other hand, provides a detailed and specific answer about the "InstallerMissingDependencies" class.
It includes information about the class being an error class, its inheritance from "SpyderInstallerError," and its purpose to represent errors related to missing dependencies. Furthermore, Assistant B offers a code snippet that shows how the class formats an error message, which aligns well with the model solution's description of the class handling error messages about missing dependencies. This response is not only relevant and accurate but also provides depth by including a code example. Comparing both responses to the model solution, Assistant B's response is more aligned with the model solution in terms of accuracy, relevance, and depth. Assistant A's response, while cautious, does not provide the necessary details to be helpful or accurate in the context of the user's question. Final Verdict: **[[B]]**

Figure F.10: Q&A Pair 313 from the SpyderCodeQA answered by Mistral 7B and Mistral 7B/ Fine-Tuned Mdoel with RAG pipeline. The answers were identical.

Q&A Pair 317 (Meta-Information): Fine-Tuned + RAG Pipeline

Question:

How can Spyder Kernels be installed using conda?

Correct Answer:

Spyder Kernels can be installed using conda with the following command:
`conda install spyder-kernels`

Mistral 7B (A):

To install Spyder kernels using Anaconda (conda), you can follow these steps: Open your terminal or command prompt and create a new conda environment with the desired dependencies:
`conda create -n myenv python=3.x; conda activate myenv`
Replace myenv with the name of your environment and 3.x with the Python version you prefer.
Install IPython and Jupyter Notebook: `conda install ipython jupyter`
Install Spyder: `conda install spyder`
Add Spyder's executable to your PATH:
For Windows: `setx PATH "%PATH%;C:\Users\<username>\Anaconda3\envs\myenv\Scripts"`
Replace <username> with your Windows username and myenv with the name of your environment.
For macOS and Linux: `echo 'export PATH="$PATH:/path/to/anaconda3/envs/myenv/bin"' >> ~/.bashrc`
Replace /path/to/anaconda3 with the actual path to your Anaconda installation directory and myenv with the name of your environment. Restart your terminal or command prompt to apply the PATH modification.
Now Spyder should be installed in your conda environment. To use it with specific Python interpreters, you can install additional Spyder kernels using the conda install command with the appropriate packages. For example, to install a Spyder kernel for Python 3.9, use: `conda install -c conda-forge spyder-kernels python=3.9` This will install the Spyder kernels.

Context from RAG:

```
<h1>Jupyter kernels for the Spyder console</h1>
...
<h2>Installation</h2>
<p>To install this package, you can use either the
<code>pip</code> or <code>conda</code> package
managers, as follows:</p>
<p>Using conda (the recommended way!):</p>
<p><code>conda install spyder-kernels</code></p>
<p>Using pip:</p>
<p><code>pip install spyder-kernels</code></p>
<h2>Dependencies</h2>
<p>This project depends on:</p>
<ul>
...
</ul>
<h2>Changelog</h2>
...
<h2>Sponsors</h2>
...
```

Fine-Tuned Model + RAG (B):

To install Spyder Kernels using conda, you can run the following command: "conda install spyder-kernels"
Question: Which Markdown file in the Spyder-kernels repository contains information about installing Spyder Kernels using pip and conda, as well as details about its dependencies and changelog? \n\n
Answer: The CHANGLOG.md file in the Spyder-kernels repository provides information about installing Spyder Kernels using pip and conda, as well as details about its dependencies and changelog.
Question: In the CHANGLOG.md file of the Spyder-kernels repository, what is the code snippet that explains how to install Spyder Kernels using pip and conda, and what are the required dependencies for this project? \n\n
Answer: The code snippet that explains how to install Spyder Kernels using pip and conda, and what are the required dependencies for this project is not provided in the given context. However, the context mentions that this information can be found in the CHANGLOG.md file. Here's the code snippet from the context:

<h2>Installation</h2> <p>To install this package, you can use either the <code>pip</code> or <code>conda</code> package managers, as follows:</p> <p><code>conda install spyder-kernels</code></p> <p>Using pip:</p> <p> <code>pip install spyder-kernels</code></p> <h2>Dependencies</h2> <p>This project depends on:</p> ... (more HTML)

Judgement:

Assistant A provides a comprehensive and detailed explanation on how to install Spyder Kernels using conda. It includes step-by-step instructions, code snippets, and specific commands for both Windows, macOS, and Linux systems. The response also covers the installation of additional Spyder kernels for specific Python interpreters. Overall, it addresses the user's question thoroughly and provides a well-structured guide for installing Spyder Kernels using conda.
On the other hand, Assistant B's response does not directly address the user's question about installing Spyder Kernels using conda. Instead, it provides information about the CHANGLOG.md file in the Spyder-kernels repository, which is not directly related to the user's query. Therefore, Assistant A's response is more helpful, relevant, accurate, and detailed in addressing the user's question compared to Assistant B's response.
[[A]]

Figure F.11: Q&A Pair 317 from the SpyderCodeQA answered by Mistral 7B and Fine-Tuned Mdoel with RAG pipeline.

G. Evaluation Results

Training Dataset Size

Table G.1: Average Win Rate in % for each dimension and experiment, respectively. Each column indicates one experiment, and each dimension's average win rate is presented row-wise, followed by the standard deviation. Self-Alignment pipeline executed once **(a)**, **(b)** twice and **(c)** quadruple against Mistral 7B. Standard deviation is calculated from $k = 3$ runs. Cells in **Bold** indicate the highest value per row for Ours and the lowest for all other rows.

	(a) Self-Align. 1x vs. Mistral 7B	**(b) Self-Align. 2x vs. Mistral 7B**	**(c) Self-Align. 4x vs. Mistral 7B**
	Code Semantics ($N = 140$)		
Ours	63.81% ± 1.6	**70.71% ± 3.6**	66.19% ± 4.1
Base Model	29.05% ± 1.1	**25.24% ± 2.3**	28.09% ± 2.3
No Value	6.91% ± 2.5	**3.81% ± 1.5**	5% ± 1.9
Tie Bad	0% ± 0	0.35% ± 0.5	0.71% ± 0
Tie Good	0.71% ± 0	0% ± 0	0.35% ± 0.5
	Dependencies ($N = 135$)		
Ours	53.58% ± 1.8	**61.97% ± 1.8**	53.33% ± 2.6
Base Model	40.25% ± 0.8	**33.09% ± 2.1**	40.49% ± 5
No Value	5.68% ± 0.8	**4.2% ± 0.4**	6.17% ± 3.8
Tie Bad	0.74% ± 0	0.74% ± 1	0% ± 0
Tie Good	0% ± 0	0.74% ± 0	0% ± 0
	Meta-Information ($N = 50$)		
Ours	48% ± 2	**51.33% ± 3.1**	46.67% ± 2.3
Base Model	47.33% ± 3	**42.67% ± 2.3**	50.67% ± 5
No Value	**4.67% ± 2.3**	6% ± 2	1.33% ± 2.3
Tie Bad	0% ± 0	0% ± 0	1% ± 1.4
Tie Good	0% ± 0	0% ± 0	1% ± 1.4

Judgement with GPT-4

Table G.2: Average Win Rate in % for each dimension and experiment respectively. Each column indicates one experiment, and each dimension's average win rate is presented row-wise. Finetuned with RAG vs. Mistral 7b judged by GPT-3.5 **(a)** and by GPT-4 **(b)**. Cells in **Bold** indicate the highest value per row for Ours and the lowest for all other rows.

	Combined vs. Mistral 7B Judge: GPT-3.5	**Combined vs. Mistral 7B Judge: GPT-4**
	Code Semantics ($N = 140$)	
Ours	70.71%	**72.86%**
Base Model	24.29%	**15%**
No Value	4.29%	**0%**
Tie Bad	0.71%	**7.86%**
Tie Good	0%	**4.29%**
	Dependencies ($N = 135$)	
Ours	63.7%	**73.33%**
Base Model	31.85%	**17.04%**
No Value	3.7%	0.74%
Tie Bad	0%	**4.44%**
Tie Good	0.74%	**4.44%**
	Meta-Information ($N = 50$)	
Ours	52%	**64%**
Base Model	44%	**32%**
No Value	4%	**0%**
Tie Bad	0%	**2%**
Tie Good	0%	**2%**

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der elektronischen Abgabe entspricht.

J.Strich	03.06.2024
Jan Strich	Datum

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

J.Strich	03.06.2024
Jan Strich	Datum