UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE POLITICHE,
ECONOMICHE E SOCIALI

Master's Degree in Data Science for Economics

Generative AI for Automated Legacy Mainframe Documentation

Supervisor:
Prof. Alfio FERRARA

Co-supervisor:
Prof. Stefano MONTANELLI

External Co-supervisor:
Dr. Enrico PAPALINI

Candidate:
Riccardo STURLA
13457A

ACADEMIC YEAR 2023/2024

Acknowledgments

Abstract

Legacy mainframe systems modernization has become a critical need for many organizations seeking to enhance their operational efficiency and scalability. Despite their robustness and reliability, these legacy systems often suffer from limitations in agility, integration capabilities and maintenance expenses, posing significant challenges to businesses aiming to stay competitive. Euronext Securities Milan is exploring the potential of generative AI tools in addressing these challenges by facilitating the transformation of outdated mainframe codebases and architectures into modern and understandable ones. This thesis aims to demonstrate how large language models (LLMs) can automate code understanding and refactoring, starting with the production of functional documentation. In particular, after an introduction about the current landscape of LLMs and generative AI for coding, a wide set of solutions are explored, tested and evaluated. Closed-source and open-source models were employed and compared to see how they perform on the task and to evaluate whether their performances can match. Significant effort was invested in prompt engineering to optimize the model's outputs. Finally, an evaluation framework was established to assess the results, focusing on metrics such completeness, clarity and consistency. Through the comprehensive analysis of case studies and implementation strategies described, we were able to prove the pivotal role that AI can cover in this kind of tasks, ensuring long-term sustainability and innovation in the enterprise landscape.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This master's thesis is based on the work done during my internship in Euronext Securities Milan (ESM), under the supervision of Enrico Papalini, from April to October 2024.

Monte Titoli, owned by Borsa Italiana, is the central securitities depository (CSD) for the Italian financial market and one of the biggest in Europe. In November 2021 Monte Titoli was renamed Euronext Securities Milan (ESM) after the merging in Euronext Securities, the pan-European service for CSDs of the Euronext group. ESM offers settlement and custody services on Italian and foreign securities on over 40,000 financial instruments including bonds, government securities, debt securities, ETFs, and certificates.

During the internship, we worked on the modernization of legacy mainframes through the implementation of generative AI tools and techniques. In the next introductory pages, we will formalize the problem we are trying to solve, describe the subjects covered in the thesis, overlook the available infrastructure and explore the state of the art in the world of LLMs.

1.1 Motivation and Problem Formalization

> It is our job to create computing technology such that nobody has to program and that the programming language is human. Everybody in the world is now a programmer. This is the miracle of artificial intelligence.
>
> - Jensen Huang - *CEO of Nvidia*

Jensen Huang's speech, during the Word Government Summit in Dubai (February 24, 2024), encloses the potential of AI in democratizing technology. As artificial intelligence continues to evolve, it increasingly narrows the gap between human intention and machine execution, making complex programming tasks accessible to everyone. The shift towards natural language processing and user-friendly AI models leads to a new era where the barriers to coding are dismantled, allowing individuals to exploit the possibility of technology without the need for traditional programming skills. In this thesis, we are going to explore the innovative solutions and methodologies that are driving this paradigm shift, focusing on how Large Language Models can revolutionize code understanding and refactoring in the process of mainframe modernization.

Most mainframe applications are written in old programming languages such as COBOL or other IBM languages, which are pillars of programming in the banking and insurance sectors. These languages are no longer taught in schools and universities, making the replacement of experts complicated and costly. Nowadays, there is an acute knowledge gap as key developers have retired or will retire soon, and the void of application expertise could drive risky decisions with unpredictable outcomes.

> "Knowledge is leaving the building and no documentation is left behind.
> It seems like this is starting to be a common problem around the world."
>
> *– Bill Hinshaw, Founder COBOL Cowboys, October 2020*

Developers are forced to spend most of their time trying to understand how a single application works. In this context, AI can significantly speed up the process by providing automated tools for code comprehension, documentation, and even code generation. Our goal is to demonstrate that, through Generative AI, it is possible to bridge the knowledge gap and maintain these critical systems more efficiently. Euronext is doing its first step towards this process, starting by the understanding of how the current legacy applications work, which of them are still useful and which of them will be part of the final migration. In this dissertation, we are going to show how to automatically produce complete, accurate and understandable code documentation using the best proprietary AI models on the market; then we are going to work on how to achieve similar results with open-source solutions, overcoming obstacles and difficulties; finally we are going to explore further possible use cases to fully exploit generative AI capabilities in Euronext Securities.

1.2 Subjects covered

The specific subjects covered by this dissertation are:

- **Chapter 1** - Introduction: motivation, data considered, work environment and state of the art.
- **Chapter 2** - Evaluation framework: dissertation about techniques to evaluate LLMs output.
- **Chapter 3** - Prompt engineering: definition of prompt engineering, most common and efficient prompting techniques, prompts for the specific use case.
- **Chapter 4** - Exploring solutions: experiments carried out and their evaluation and comparison.
- **Chapter 5** - Further implementations: enhancement of the documentation, RAG, and Fine-Tuning

- **Chapter 6** - Conclusion

1.3 Available Data

For privacy reasons, the analyzed programs and applications cannot be fully reproduced here, but their characteristics and functionalities will be described in detail. The scope of experimentation and analysis extends across three different programming languages: COBOL, EGL, and PL/1. The applications considered contain programs of varying complexity and length, in terms of lines of code (LOC) or tokens, together with additional peculiarities that will be presented in the following sections.

1.3.1 COBOL Application

Common Business-Oriented Language (COBOL) is a programming language designed in 1959 for business and widely used in mainframe applications. Most of the Euronext Securities Milan codebases are written in COBOL, and its understanding represents the main need and motivation for the experiment carried out. Consequently, the analysis will focus primarily on this specific language. Nowadays, fewer people study and understand COBOL, and new functionalities are added, where possible, through the use of other programming languages. Hence, the necessity to make the functioning of old but still used applications understandable to any analyst who needs it, without requiring a deep knowledge of COBOL.
The specific application analyzed during our experiments is a financial securities management system, handling stripping and related operations, and its architecture is composed of the following main components:

1. COBOL transactional programs WSOC83 and ARMP3025
    - Handle the processing of incoming requests and messages
    - Interact with queues and tables to read or write data

- Perform formal and application checks on the data
- Update system tables based on the requested operations
- Manage anomalies and error reporting

2. Utility programs SYSPTAB5, SYSPUT06, and UTIP0060
   - Provide support functions such as table lookup, string and date conversion
   - Are called by transactional programs when necessary
3. COBOL COPY (e.g., ARE33I, ARET101, WSC01I, WSC83I)
   - Contain data structure definitions and record layouts
   - Are included in COBOL programs to facilitate data manipulation

The application is overall composed of 7000 lines of code, with the main program (WSOC83) being around 3800 LOC. The code presents in-line comments, written in Italian, which is a factor that will be considered during the following analysis.

1.3.2 EGL Program

Enterprise Generation Language (EGL) is a high-level, versatile programming language designed by IBM. It is intended to simplify the development of enterprise applications by abstracting the complexities of underlying technologies. EGL allows developers to write applications that can run on multiple platforms, such as Java, Java EE, and COBOL, and can be deployed on various environments including mainframes, web, and cloud. One of the distinctive features of EGL is, in fact, its ability to generate code in other languages, making it highly adaptable for different runtime environments and legacy systems.

EGL is vastly used by Euronext Securities Copenhagen, therefore, to extend the scope of our experiments to other CSDs in the group that might utilize our application and methods in the future, we also analyzed some programs provided by our Danish colleagues.

The EGL code analyzed is a batch program named "BMngmegaraExportHolding-Info" (alias B2017051) that is responsible for exporting holding information from the financial system to an external system called CA4U. The program follows a standard structure for batch processing, including initialization, main processing, and exception handling and has a length of 1000 LOC. The key capabilities and features of the entire application include:

- Exporting holding information based on two run types: *Complete* (all holdings) and *Delta* (changes since the last run)
- Retrieving holding data from database tables
- Processing holding records to calculate quantities and prepare data for export
- Sending the processed holding data to CA4U via MQ messages
- Supporting performance testing and the ability to include zero quantity holdings
- Implementing restart capability based on ISIN, participant identifier, and account number

The program has in-line comments, written in Danish.

1.3.3 PL/1 Programs

Programming Language One (PL/1) is a high-level programming language developed by IBM in the early 1960s. It was designed to combine the features of scientific, engineering, and business programming languages. Euronext Securities Oslo still has some of their applications written in PL/1, so we decided to carry out some tests on four batch programs:

- AE20H012: handles the transfer of subscription rights holdings. Its main purpose is to move holdings to allocation information, update the main ledger, and generate necessary output files for further processing.

- AE26H02: handles the deletion of old shares after a merger or exchange. It processes share information, updates relevant databases, and generates necessary output files for further processing.
- AE26H013: responsible for handling the exchange of allocated continuous securities. It processes securities that are undergoing various corporate actions such as mergers, demergers, and bonus issues.
- AT02H01: handles the process of shifting share fund codes in the financial system. Its primary purpose is to update share holdings, change share fund codes, and update the share ledger and the date of change in a specific external table.

All the programs have in-line comments, written in Norwegian.

1.4 Infrastructure and Work Environment

In this section, we review the infrastructure that has been used to conduct the analysis.

1.4.1 Cloud and AWS

1.4.1.1 Amazon EC2

Amazon Elastic Compute Cloud (Amazon EC2) is AWS cloud compute platform, which offer a broad set of on-demand VMs. The first work environment chosen for the project was a pre-existent Amazon Web Services EC2 instance, g4dn.8xlarge, with NVIDIA T4 GPU, 32 vCPUs, 128.0 GiB of memory and 50 Gibps of bandwidth. The instance has a cost of 1.37$/h of usage and was already used for the developing of other applications in ESM.

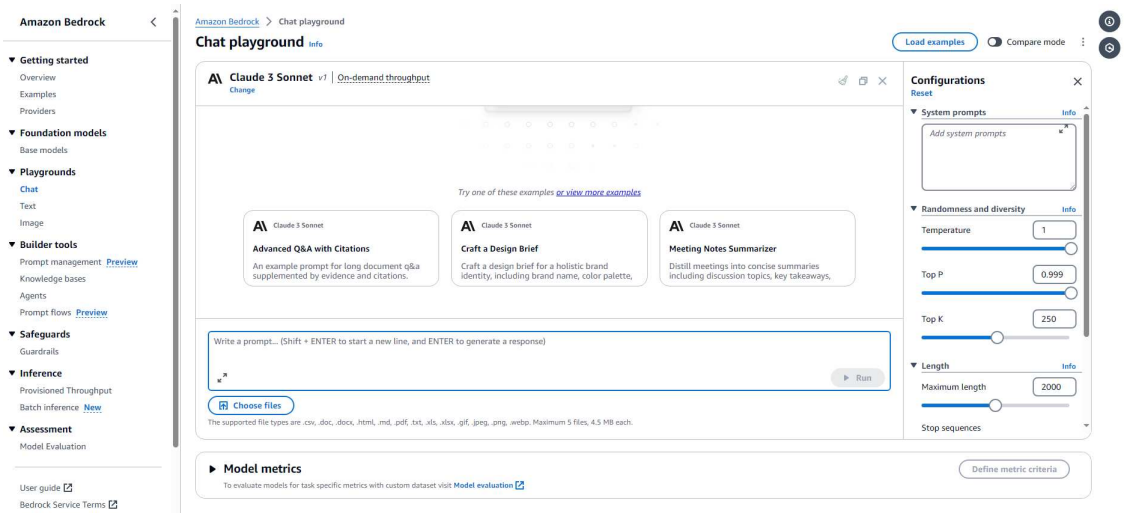

Figure 1.1: Amazon Bedrock - User Interface and Chat playground

1.4.1.2 Amazon Bedrock and Sagemaker

During my internship, I had the opportunity to test and work with AWS services Bedrock and SageMaker. Amazon SageMaker is a fully managed service for machine learning, with tools for any use case. SageMaker enables the possibility to build, train and deploy ML models through convenient notebook environments and other tools integrated into a practical IDE. Amazon Bedrock, on the other hand, is AWS's new tool that makes high-performing foundation models from AI companies available through a unified API. Bedrock also offers a convenient environment accessible via UI for experimenting with models and prompts, as well as tools for creating agents or evaluating outputs (Figure 1.1). It is possible to use the Bedrock API to make calls to LLMs (Bedrock offers a wide selection, both free and on-demand) in SageMaker notebooks and thus begin creating generative AI applications entirely within AWS.

1.4.2 Local Environment

Together with Amazon Cloud Services, a parallel environment was set up on a local laptop. The pc is a DELL XPS 15 with 13th Gen Intel Core i9-13900H 2.60 GHz processor, 32GB RAM and a NVIDIA GeForce RTX 4070 GPU. The laptop has a Windows Subsystem for Linux (WSL), running Linux distribution Ubuntu. WSL

is a feature of Windows that allows to run a Linux environment on a Windows machine, without the need for a separate virtual machine or dual booting.

1.4.2.1 LangChain

LangChain is an open-source framework designed to simplify the development of applications powered by large language models. It provides a suite of tools and components that help developers build, deploy, and manage LLM applications efficiently [25]. It is used for chaining the end-to-end different pieces of any Gen AI app (retrieval, generation, etc.). It comes under the suite of a python library and it was utilized in our scripts to integrate useful features such as chains and ad hoc splitters. Langchain and its functionalities were widely use in the developing of our applications.

1.4.2.2 Ollama

Ollama is a tool for running and customizing large language models on local machine. It offers a library of the vast majority of open-source models and it is integrated with a variety of tools, frameworks, and programming languages (Python, LangChain, LlamaIndex, etc.), making it easy to incorporate LLMs and AI in applications. We used Ollama combined with LangChain to develop our solutions in the best way possible.

1.5 State of the Art

Large Language Models are powerful AI tools designed to understand and generate human language. They are capable of a wide range of natural language processing tasks and represent one of the most revolutionizing technology of recent years. LLM are typically based on neural network transformer architecture and are trained on huge amounts of data. The "large" part refers to the fact that these models have a very large number of parameters, often in a range of hundreds of billions.

In this section, we will provide an overview of the LLM landscape, highlighting both closed-source and open-source alternatives available on the market. We will discuss their strengths and weaknesses, exploring potential future trends and their impact on the world of coding.

1.5.1 Closed-source models

Closed-source LLMs are models developed by organizations that do not make their underlying code, weights, and training data, available. By keeping the model closed-source, companies can monetize their LLMs through paid APIs or subscription services. This is a common business model for companies like OpenAI with GPT or Anthropic with Claude. Moreover, such companies can retain full control over the usage and the training of their models. Naturally, with the global diffusion of such tools, numerous ethical questions arise. Legal frameworks and debates surrounding these issues are still very much ongoing and evolving.

In recent years, the world of generative AI has escalated with the progressive release of increasingly powerful and large models. Leading this growth are four major companies: OpenAI, Anthropic, Google, and Meta. However, Meta stands out by releasing its models in open-weight format. As shown in Figure 1.2, the closed-source landscape has intensified in recent months with the release of the two models currently considered the most powerful: Claude 3.5 Sonnet and GPT-4o. These models achieve the highest average benchmark scores on the market, with Claude leading the rankings. These models, in addition to their incredible performance and generative capabilities, including multimodal abilities, feature a very large context window (Figure 1.6). This represents a significant advantage in analyzing very long textual documents, such as in our specific use case. As we will see in the following chapters, we have chosen Claude 3.5 Sonnet as the closed-source alternative for developing our tests and evaluations. This model has proven to be, as claimed by Anthropic, incredibly adept at understanding code and generating well-structured and complex textual documents, including the integration of graphical elements and

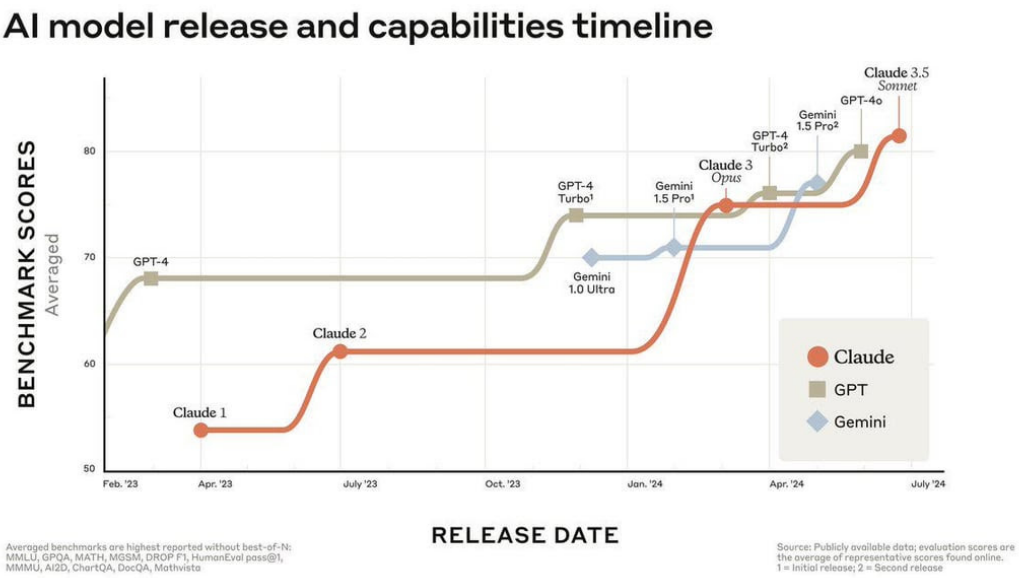

Figure 1.2: AI model release and capabilities timeline

tables. Some external providers have proposed solutions leveraging GPT-4o with excellent results. The choice between the two is not based on the quality of the produced documentation but rather on the larger context window of Claude, which ensures greater reliability in managing any type of input and minimizes the risk of output truncation due to token limits.

1.5.2 Open-source models

Open-source large language models are LLMs that are publicly available for anyone to use, customize, and distribute, allowing any person or business to use them for their means without having to pay licensing fees. Unlike their closed-source counterparts, open source LLMs provide (not always) full access to their underlying code, training data, and model architecture. This transparency creates a collaborative environment where developers and researchers can innovate and improve the models continuously. One of the main advantages of open source LLMs is their cost-effectiveness: by eliminating licensing fees associated with proprietary models, companies can significantly reduce expenses. Additionally, the flexibility to customize and fine-tune these models to meet specific needs is a major benefit, allowing for

customized solutions that can be adapted to unique use cases. However, the quality and performance of these models can vary widely, as they depend on the resources and expertise of the community maintaining them. Furthermore, implementing and managing open-source solutions often requires a higher level of technical expertise, which can be a barrier for some organizations. Despite these challenges, we believe the open source approach offers a powerful alternative for leveraging the capabilities of LLMs and, balancing the advantages of cost savings and customization with the potential drawbacks, organizations can make informed decisions about integrating open source LLMs into their workflows.

The early developments in the landscape of open-source LLMs saw the emergence of significant base models such as the first version of BLOOM and OPT. However, it was with the advent of Llama that the open-source paradigm managed to achieve high and comparable performances to closed-source benchmarks like GPT-3. Llama sparked an explosion in open-source research: after its weights were made public, the community began to develop a wide range of derivative products and fine-tuned versions. Since then, research and experimentation have not stopped, and today

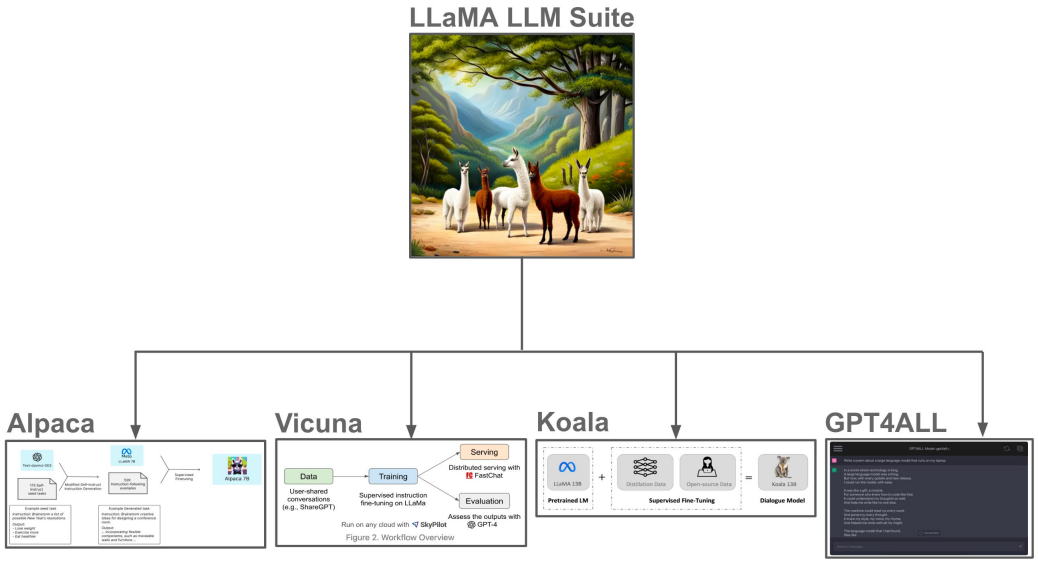

Figure 1.3: Llama Suite [42]

there are thousands of open-source products available and runnable locally. Meta

Llama 3 is the latest releases in the Llama family. Published on 18 April 2024, Llama 3 shows state-of-the-art performances on a wide range of industry benchmarks. Its two version, 7B and 80B, are claimed to be "the best open source models of their class" [1]. The biggest version, Llama 3.1 405B, has just concluded the training process and over the coming months, multiple models with new capabilities including multimodality, the ability to converse in multiple languages, a much longer context window, and stronger overall capabilities are expected to be released by Meta [2].

It is impossible not to mention another state-of-art LLM like Mistral, with all its derivatives. Mistral AI is a French company founded in April 2023 and specialized in artificial intelligence products. The first model they released was Mistral 7B, in September 2023, then Mixtral 8x7B, their first sparse mixture-of-experts released December 2023 and, ultimately, Mixtral 8x22B, Mistral AI's best open source model released April 2024. Mixtral 8x22B has the following capabilities: [3]

- It handles a context of 32k tokens.
- It handles English, French, Italian, German and Spanish.
- It shows strong performance in code generation.
- It can be finetuned into an instruction-following model with great benchmark performances.

Alongside Llama and Mistral, we can find BLOOM, Gemma, OPT, GPT-NeoX, Falcon, Bert, and many others. The landscape is vibrant, with continuous new releases and versioning. Despite the closed-source models not seeming to slow down their growth trajectory, the open-source scenery is keeping pace and continues to exploit its peculiarities and advantages to gain ground, especially in the development of AI applications within companies.

[1] Introducing Meta Llama 3: The most capable openly available LLM to date, Meta Blog, 18 April 2024

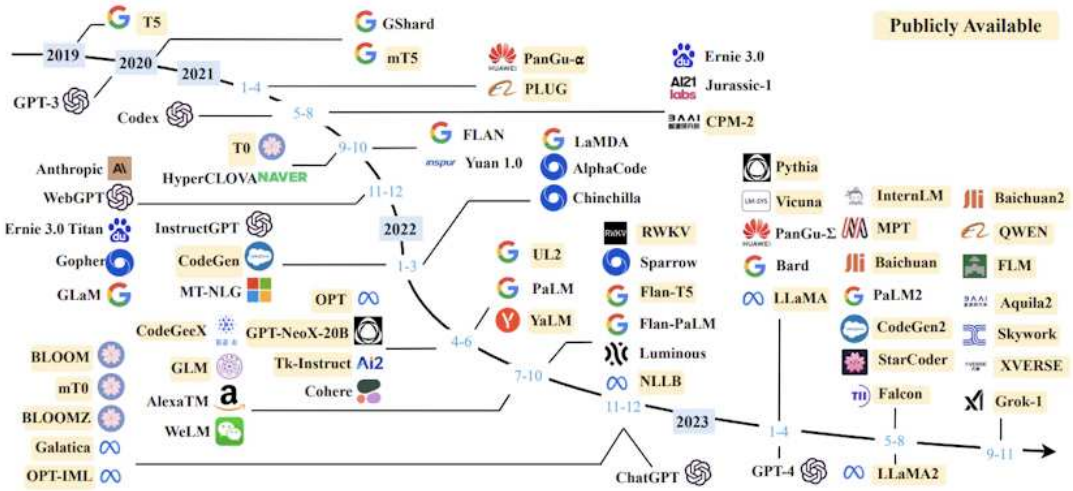

Figure 1.4: A timeline of existing large language models (having size larger than 10B) in recent years, up to December 2023. LLMs with publicly available model checkpoints are highlighted in yellow [17]

1.5.3 Coding

Language models are significantly impacting the world of programming, and this thesis is based precisely on this premise. From code autocompletion to its complete generation, including its review and translation, AI is already assisting millions of developers in their daily tasks. We are not yet at the point referenced by Jensen Huang (Section 1.1), but as with everything related to AI, the future possibilities seem endless, and progress is rapid and unstoppable. In particular, coding assistants have become essential in companies which code and develop internally. For instance, Euronext Securities Milan recently concluded a testing period of a generative AI tool for code assistance and writing, GitHub Copilot. GitHub Copilot is an AI coding assistant that helps write code faster and with less effort. Copilot offers coding suggestions while typing, directly in the IDE, both completion of the current line or whole new block of code. Internal analytics and surveys based on the initial stage of the coding assistant integration in ESM highlight 15-20% of cost savings and 11-15% of productivity improvement. Specifically, AI Integration Impact on a 4000 Man-Day Project has resulted in:

1. **Cost Savings:** Estimated about 600 man-days saved.

2. **Productivity Gain:** Additional 440 man-days worth of work.

This means that not only that Copilot could potentially save about 600 man-days in costs, but it could also gain an additional 440 man-days worth of productivity in the same timeframe. Copilot business has a cost of $19 per developer per month or $228 per year, with about 10.000€ estimated savings per resource. After the phase of testing, Euronext Securities Milan is planning to fully adopt GitHub Copilot in Q3 2024. This was just an example of how AI has already permeated the corporate environment, especially in the realm of coding assistants.

Multiple LLMs are trained or fine-tuned to perform coding tasks; the most relevant are shown in Fig. 1.5 and 1.6. All are generally very effective for completion, generation, and review tasks, but the specific use case of this thesis requires particular features that currently only a few possess. The context window (CW) of a model refers to the amount of text the model can process at once when generating or understanding language. This window is measured in terms of tokens (words or parts of words) and directly influences the information the model can leverage for subsequent token predictions. A larger context window allows for a deeper understanding of the text and more coherent responses, but it also increases computational demands. Models must strike a balance between context and efficiency to be effective. In the analysis of very long documents or programs, as in our case (Section 1.3), a large context window, as we will see, is fundamental in ensuring efficiency and quality of the output produced. As shown in Fig. 1.5 and 1.6, while the best proprietary models all have very large CWs (>100k tokens), the open-source landscape has only recently expanded its catalog with models featuring a larger base CW. Despite the existence of methods to extend the context of the models [11], these require a greater amount of work and skills and do not guarantee the maintenance of the original performance by the LLMs. However, the general trend seems to be moving towards LLMs with increasingly longer contexts on both fronts.

After a first sorting, the open-source models which we decide to compare and

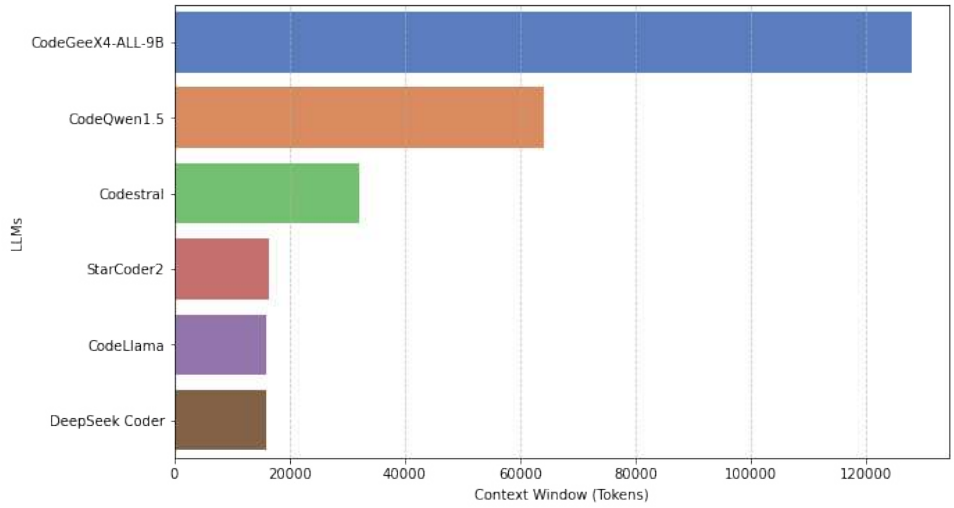

Figure 1.5: Open-source Context Windows (tokens)

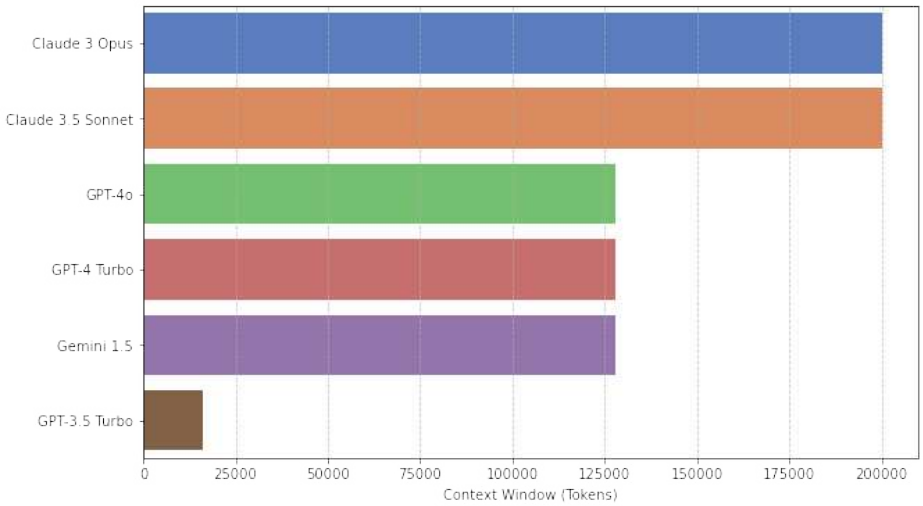

Figure 1.6: Closed-source Context Windows (tokens)

analyse during the experiments for code documentation generation are four:

- Code Llama.

  Code Llama is a code-specialized version of Llama 2 that was created by further training Llama 2 on its code-specific datasets, sampling more data from that same dataset for longer. Essentially, Code Llama features enhanced coding capabilities, built on top of Llama 2. It can generate code, and natural language about code, from both code and natural language prompts. It can also be used for code completion and debugging. It supports many of the most popular

languages being used today, including Python, C++, Java, PHP, Typescript (Javascript), C, and Bash. There exists four sizes of Code Llama with 7B, 13B, 34B, and 70B parameters respectively. Each of these models is trained with 500B tokens of code and code-related data, apart from 70B, which is trained on 1T tokens. The 7B and 13B base and instruct models have also been trained with fill-in-the-middle (FIM) capability, allowing them to insert code into existing code, meaning they can support tasks like code completion right out of the box. [1] Code Llama 7B-instruct was the specific version of the model we chose to test, due to memory and size reasons.

- CodeStral.

  CodeStral is the newest release of Mistral AI and one of the latest model for coding in general. Codestral is trained on a diverse dataset of 80+ programming languages, including the most popular ones, such as Python, Java, C, C++, JavaScript, and Bash. It also performs well on more specific ones like Swift and Fortran. This broad language base ensures Codestral can assist developers in various coding environments and projects. [4]

- CodeQwen1.5 7B - Chat.

  CodeQwen is based on Qwen1.5, a language model series, built by Alibaba Cloud, including decoder language models of different model sizes. It is trained on 3 trillion tokens of data of codes. It demonstrated strong code generation capabilities and competitve performance across a series of benchmarks; it supports long context understanding and generation with the context length of 64K tokens; it supports 92 coding languages; it shows excellent performance in text-to-SQL, bug fix, etc. [7]

- CodeGeeX4-ALL-9B.

  CodeGeeX4-ALL-9B is the latest open-source version of CodeGeeX4 model series. It is a multilingual code generation model that can support comprehensive

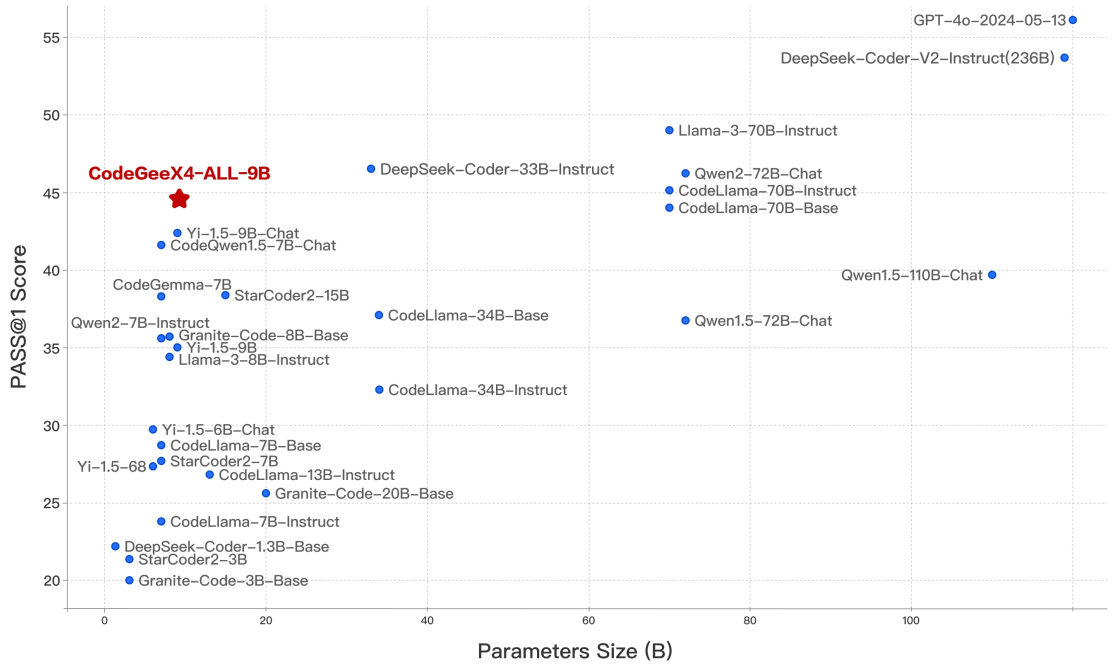

Figure 1.7: Scores of LLMs on BigCodeBench benchmark, filtered by parameters size [49]

functions such as code completion and generation, code interpreter, function call, and repository-level code Q&A. CodeGeeX4-ALL-9B is trained on 23 programming language (way less than CodeQwen1.5), however, it has achieved highly competitive performance on public benchmarks, such as BigCodeBench (Figure 1.7) and NaturalCodeBench. It is currently the most powerful code generation model with less than 10B parameters, even surpassing much larger general-purpose models, achieving the best balance in terms of inference speed and model performance. Moreover, CodeGeeX4 has a context window of 128k tokens, making it a potentially valuable choice for our use case. [49]

The choice of models with such a small number of parameters is due to hardware limitations. Running the LLMs locally requires an amount of RAM memory proportional to the number of parameters.

1.5.4 Open-source vs. Closed-source

Proprietary closed-source models are generally more powerful than their open-source counterparts and their progression in terms of performances has been remarkably rapid over the last year (Fig. 1.2); however, they are undoubtedly more expensive. The companies that own these models usually provide access via APIs at a cost based on input and output tokens. These costs need to be evaluated from a business perspective through appropriate analysis, examining potential usage and trade-offs. Another critical consideration for companies like Euronext concerns compliance and data privacy. When using LLMs that are not deployed locally, it is essential to consider where and how prompts containing potentially sensitive and protected company information are processed. Data management depends on the individual provider, the geographic location and any specific contracts in place. Generally, using prompts for training and/or improving models is the standard when API calls are made directly through the model provider.

A different scenario applies to providers who have partnership with the companies owning the LLMs and offer these models through their platforms. For instance, at Euronext, the use of Amazon Bedrock has been largely discussed. As said in Chapter 1, Amazon Bedrock is AWS's fully-managed service that offers a wide range of foundation models to integrate into applications and combine with other AWS tools. In this case, the data usage policies are those of AWS rather than the AI companies (e.g., Anthropic). Bedrock is in scope for common compliance standards such as Service and Organization Control (SOC), International Organization for Standardization (ISO), Health Insurance Portability and Accountability Act (HIPAA) eligibility, and customers can use Bedrock in compliance with the General Data Protection Regulation (GDPR). Moreover, "*Amazon Bedrock doesn't store or log your prompts and completions. Amazon Bedrock doesn't use your prompts and completions to train any AWS models and doesn't distribute them to third parties. The model providers don't have access to the AWS accounts and they don't have access*

to Amazon Bedrock logs or to customer prompts and completions." [2]

Therefore, there are various considerations to make and, especially if there are already relationships and familiarity with the provider in question, the evaluation could be simpler and more immediate. Clearly, none of these issues arise when using local open-source models, where everything is done internally without transferring documents and prompts. On the other hand, developing internal solutions can require a deeper technical knowledge, that may not always be available within the company, and sufficient powerful hardware.

[2] Amazon Bedrock Security and Compliance page. [39]

Chapter 2

Evaluation framework

Comprehensively assess LLMs' effectiveness and accuracy is a crucial aspect of understanding and improving their performances. Unlike traditional machine learning models, where evaluation metrics are often straightforward, the assessment of LLMs involves more "abstract" criteria due to the complexity and variability of natural language. Task-specific metrics are often employed depending on the application, such as BLEU scores or ROUGE scores. However, human evaluation remains a crucial component in the evaluation process: while automated metrics provide a quick and objective way to evaluate certain aspects of models' performance, human judges are often needed to assess other qualities which are difficult to quantify algorithmically. Moreover, the emerging of LLM evaluation techniques offers a wider set of choices and allows faster and potentially iterative evaluation for continuous improvement of the output. Analyzing the model's errors and imperfections we can adjust the application's architecture and improve prompt engineering techniques to enhance performances. This comprehensive approach to evaluation ensures that LLMs are not only proficient but also aligned with human expectations. In this chapter, we are going to explore the types of outputs produced and the metrics we chose to use for their assessment, establishing a robust evaluation framework through which we can judge the LLMs' behavior in the most complete way possible.

2.1 Types of documentation

To evaluate our specific use case's outputs and conduct a complete analysis of the LLMs' capabilities in generating code documentation, we decided to consider two different categories of document:

- Functional documentation
- Functional documentation and decision making

The functional documentation is the core of our experiments and represents the main goal of the generative AI project in Euronext Securities. A functional documentation should provide a complete description of the behavior of the code from a user's perspective. It should include the explanation of the purpose of the program, its functionalities, use cases, requirements and error handling, together with flowcharts and diagrams for a better understanding of the program flow and data interactions. On the other hand, the functional documentation and decision making is a document which provides insights of some code programs and give us suggestion for eventual rewriting or dismission of applications and functionalities. Decision making is a side task that we decided to test on a subset of codes (Pl/1 programs, see section 1.3.3) in order to explore and expand the possibilities of our solution and it will be discussed in Chapter 5.

2.2 Evaluation metrics

In this section, we will introduce the evaluation metrics considered for assessing the outputs generated by LLMs, from automated metrics to human evaluation.

2.2.1 Overall Qualitative Evaluation - OQE

The initial evaluation criterion considered is a simple qualitative analysis of the output which we are going to call "Overall Qualitative Evaluation" (OQE). This

assessment is performed manually, involving a summary evaluation of the produced document, taking into account readability, comprehensibility, formatting errors and completeness, without deep diving into content specifics. The OQE, in addition to serving as the primary filter for solution selection, will be presented alongside the other metrics and the actual scoring evaluation.

2.2.2 Benchmark evaluation

When assessing the quality of generated documentation, having a reference point that can serve as a gold standard and comparison baseline is essential. We will employ two benchmarks, selected based on the output being evaluated:

- **Human-written documentation:**

  Pre-existing documentation authored by a human expert. If present, it will be compared with the generated outputs to assess completeness, commonalities and differences. This immediate tool allows us to evaluate whether the use of generative AI has led to satisfactory results and additional benefits beyond cost and time savings.

- **Closed-source generated documentation:**

  Documentation generated using AI during the conducted experiments. It will be used as a reference alongside the human-written documentation to evaluate the outputs produced by open-source models. This approach enables an analysis of the reliability and competence of these models for the specific task considered.

2.2.3 Automated Metrics

The automatic evaluation of natural language generation has captured researchers' attention over the past two decades and necessitates specialized metrics developed for this purpose. Numerous techniques and methods have been proposed, each

with its own peculiarities. In this section, we will analyze the two selected scores for evaluating our generated outputs: ROUGE and BERTScore. Despite these metrics providing an automated evaluation criterion based on a reference, there are several pitfalls associated with using such conventional benchmarks. First of all, they usually compare the generated text with a reference, looking for overlapping or similar elements, often neglecting the real semantic meaning of the text. Moreover, these metrics often lack adaptability across a wide variety of tasks. Applying a metric designed for one task to another is not always advisable since many metrics are designed specific for a certain purpose. Furthermore, the literature has shown that there may be a weak correlation between these metrics and human judgments, especially in tasks that require creativity and diversity. Fortunately, our use case does not demand a significant degree of creativity and inventiveness. Having in mind all the possible limitations, it is still possible to apply automated metrics in a successful manner. We are going to use them to evaluate a specific section of the produced documentation, complemented by other scores, in order to simplify and justify their application.

2.2.3.1 ROUGE (Recall-Oriented Understudy for Gisting Evaluation)

ROUGE score is one of the most used methodology for the evaluation of text generated by natural language processing (NLP) models, expecially in summarization tasks. Four methods were original proposed in the presentation paper [28] to compute such score: ROUGE-N, ROUGE-L, ROUGE-W, and ROUGE-S.

- ROUGE-N measures the overlap of n-grams (continuous sequences of n words) between the generated text and a reference text.
- ROUGE-L measures the longest common subsequence (LCS) between the generated text and a reference text. The LCS is the longest sequence of words that appear in both texts in the same order, but not necessarily contiguously.
- ROUGE-W (Weighted Longest Common Subsequence) is a variant of ROUGE-

L that gives different weights to different subsequences. It emphasizes longer contiguous sequences more than shorter or non-contiguous ones.

- ROUGE-S measures the overlap of skip-bigrams between the generated text and a reference text. Skip-bigrams are pairs of words that maintain their order but are not necessarily adjacent.

Our goal is to compare the general overview of the examined programs produced by open-source LLMs with the reference (closed-source benchmark) overview. This comparison can provide insight into the accuracy and completeness of the output, demonstrating how well the model understands the actual functioning of the program.

Given the nature of the text under examination, the most suitable metric for evaluating the output is the ROUGE-L score. This score will be complemented and supplemented by ROUGE-2 and ROUGE-1, providing insights into the basic content overlap. ROUGE-N is computed as follows:

$$\text{ROUGE-N} = \frac{\sum_{S \in \{\text{ReferenceOverview}\}} \sum_{\text{gram}_n \in S} \text{Count}_{\text{match}}(\text{gram}_n)}{\sum_{S \in \{\text{ReferenceOverview}\}} \sum_{\text{gram}_n \in S} \text{Count}(\text{gram}_n)}$$

Where n stands for the length of the n-gram, gram_n, and $\text{Count}_{\text{match}}(\text{gram}_n)$ is the maximum number of n-grams co-occurring in the candidate and in the reference overviews.

ROUGE-L requires a more complex computation and some preliminary definitions, as presented in the already mentioned presentation paper ([28]):

1. LCS: A sequence $Z = [z_1, z_2, ..., z_n]$ is a subsequence of another sequence $X = [x_1, x_2, ..., x_m]$, if there exists a strict increasing sequence $[i_1, i_2, ..., i_k]$ of indices of X such that for all $j = 1, 2, ..., k$, we have $x_{ij} = z_j$ (Cormen et al.[12]). Given two sequences X and Y, the longest common subsequence of X and Y $LCS(X, Y)$ is a common subsequence with maximum length.

2. Precision: ratio of the length of LCS to the total number of words in the

candidate text n.

$$P_{lcs} = \frac{LCS(X,Y)}{n}$$

3. Recall: ratio of the length of LCS to the total number of words in the *reference* text m.

$$R_{lcs} = \frac{LCS(X,Y)}{m}$$

4. β: A parameter that determines the relative importance of recall versus precision. Typically, β is set to 1, giving equal weight to precision and recall.

Given these definitions, it is possible to compute the ROUGE-L, which reflects how well the candidate overview matches the reference overview in terms of the longest common subsequence:

$$\text{ROUGE-L} = \frac{(1+\beta^2) \cdot \text{P}_{lcs} \cdot \text{R}_{lcs}}{\text{R}_{lcs} + \beta^2 \cdot \text{P}_{lcs}}$$

ROUGE, even if limited to the program overview, can help us with an automated metric to assess the quality of LLM's general understanding of the code. We are going to consider the F1 as overall scorer. Good ROUGE score varies depending on the task we are dealing with, but there some general guidelines to follow, even if relative comparisons are often more useful than the absolute values. For the considered ROUGE scores, the evaluation criteria are summarized in Table 2.1.

ROUGE Metric	**Good**	**Moderate**	**Bad**
ROUGE - 1	> 0.5	0.4 - 0.5	< 0.4
ROUGE - 2	> 0.4	0.2 - 0.4	< 0.2
ROUGE - L	> 0.3	0.2 - 0.3	< 0.2

Table 2.1: ROUGE Metrics assessment

2.2.3.2 BERTScore

BERTScore is a language generation evaluation metric based on pretrained BERT (Bidirectional Encoder Representations from Transformers) contextual embeddings, presented in the paper *"BERTScore: evaluating text generation with BERT"*, by Zhang et al., 2020 [47]. Unlike traditional metrics like ROUGE, which rely on exact n-gram matches, BERTScore computes the similarity of two sentences as a sum of cosine similarities between their tokens' embeddings. In this way, it is possible to account for meaning-preserving lexical and compositional diversity. Both the candidate text and the reference text are tokenized and embedded into a vector representation, then, for each token in the candidate text, BERTScore finds the most similar token in the reference text based on cosine similarity of their BERT embeddings. Similarly, for each token in the reference text, BERTScore finds the most similar token in the candidate text. Given a tokenized reference sentence $x = \langle x_1, ..., x_k \rangle$, the embedding model (BERT) generates a sequence of vectors $\langle \mathbf{x}_1, ..., \mathbf{x}_k \rangle$. Similarly, the tokenized candidate $\hat{x} = \langle \hat{x}_1, ..., \hat{x}_m \rangle$ is mapped to $\langle \hat{\mathbf{x}}_1, ..., \hat{\mathbf{x}}_i \rangle$. Precision and Recall are computed and combined in the F1 BERTScore (harmonic mean between precision and recall), similarly to what is done with ROUGE.

BERT Recall:

$$R_{\text{BERT}} = \frac{1}{|x|} \sum_{x_i \in x} \max_{\hat{x}_j \in \hat{x}} \mathbf{x}_i^T \hat{\mathbf{x}}_j$$

BERT Precision:

$$P_{\text{BERT}} = \frac{1}{|\hat{x}|} \sum_{\hat{x}_j \in \hat{x}} \max_{x_i \in x} \mathbf{x}_i^T \hat{\mathbf{x}}_j$$

BERT F1 score:

$$F_{\text{BERT}} = 2\frac{P_{\text{BERT}} \cdot R_{\text{BERT}}}{P_{\text{BERT}} + R_{\text{BERT}}}$$

As for ROUGE, we are considering the F1 as overall scorer. Good BERTScore varies depending on the task we are dealing with. After some tests, considering the

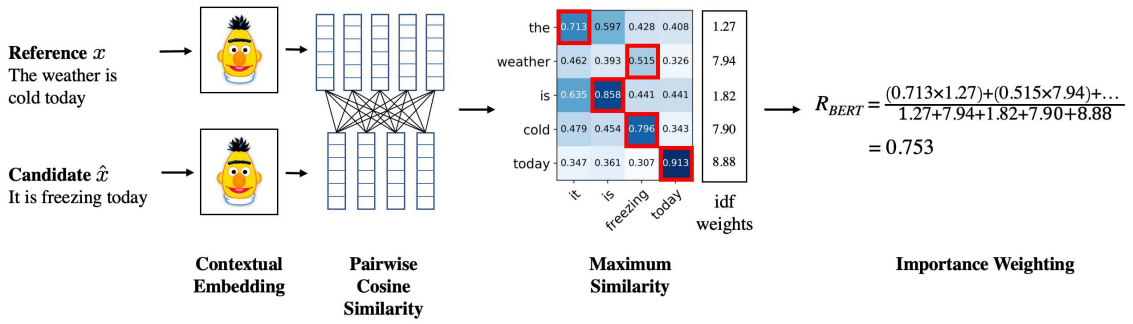

Figure 2.1: Computation of the recall metric RBERT, given the reference x and candidate $\hat{x}$, with optional idf importance weighting [47]

values for ROUGE in Table 2.1, the evaluation criteria for BERT are summarized in Table 2.2.

BERTScore		
Good	**Moderate**	**Bad**
> 0.5	0.4 - 0.5	< 0.4

Table 2.2: BERTSscore Metric assessment

We are going to use BERTScore as an additional metric for automatically evaluating the open-source LLM's general understanding of the program. This score will be applied to the overview of the documentation and combined with the other metrics to provide a complete evaluation.

2.2.4 Evaluation with LLMs

Beyond conventional evaluation metrics such as those mentioned above, an emerging trend is to use a powerful LLM as a reference-free metric to evaluate generations from other LLMs. Even if benchmarks can still be considered and utilized, gold references or human judgments are not mandatory for this kind of evaluation. G-Eval [30], Vicuna [48] and QLoRA [14] papers serve as significant examples to prove the practical effectiveness and viability of LLM assessment. By employing both simple and sophisticated prompting techniques, it is feasible to instruct a model to evaluate

an output based on predetermined criteria and scores. According to the literature, the results often closely align with human assessment.
Incorporating LLM judgment into our evaluation framework necessitated initial prompting efforts, but ultimately proved to be a valuable asset.

2.2.4.1 Selected LLM

The LLM we chose for the evaluation is one of the most proficient model available: Claude 3.5 Sonnet. Claude 3.5 Sonnet is the latest release in the Anthropic's LLMs roaster. It's Claude most powerful model, twice faster than Claude 3 Opus, with a 200k token context window. The remarkable features of Sonnet and its proven ability in code understanding make it the perfect choice for evaluating the documentation outputs. The model is given a specific prompt, detailed in the next section, which includes all the criteria to consider during the evaluation, the document to be evaluated, and the actual code to which the documentation refers. This setup enables the model to thoroughly assess the documentation, identifying any discrepancies from the original code, omissions, and errors, while also evaluating the quality of the document itself.

	Claude 3.5 Sonnet	Claude 3 Opus	GPT-4o	Gemini 1.5 Pro	Llama-400b (early snapshot)
Graduate level reasoning *GPQA, Diamond*	59.4%* 0-shot CoT	50.4% 0-shot CoT	53.6% 0-shot CoT	–	–
Undergraduate level knowledge *MMLU*	88.7%** 5-shot 88.3% 0-shot CoT	86.8% 5-shot 85.7% 0-shot CoT	– 88.7% 0-shot CoT	85.9% 5-shot –	86.1% 5-shot –
Code *HumanEval*	92.0% 0-shot	84.9% 0-shot	90.2% 0-shot	84.1% 0-shot	84.1% 0-shot
Multilingual math *MGSM*	91.6% 0-shot CoT	90.7% 0-shot CoT	90.5% 0-shot CoT	87.5% 8-shot	–
Reasoning over text *DROP, F1 score*	87.1 3-shot	83.1 3-shot	83.4 3-shot	74.9 Variable shots	83.5 3-shot Pre-trained model
Mixed evaluations *BIG-Bench-Hard*	93.1% 3-shot CoT	86.8% 3-shot CoT	–	89.2% 3-shot CoT	85.3% 3-shot CoT Pre-trained model
Math problem-solving *MATH*	71.1% 0-shot CoT	60.1% 0-shot CoT	76.6% 0-shot CoT	67.7% 4-shot	57.8% 4-shot CoT
Grade school math *GSM8K*	96.4% 0-shot CoT	95.0% 0-shot CoT	–	90.8% 11-shot	94.1% 8-shot CoT

* Claude 3.5 Sonnet scores 67.2% on 5-shot CoT GPQA with maj@32
** Claude 3.5 Sonnet scores 90.4% on MMLU with 5-shot CoT prompting

Figure 2.2: Claude 3.5 Sonnet scores on common benchmarks [5]

2.2.4.2 Prompt

Prompt for LLM evaluation

Please act as a COBOL expert, you must evaluate the documentation of the following COBOL program:

```
<code>
{{code}}
</code>
```

The documentation to evaluate:

```
<documentation>
{{documentation}}
</documentation>
```

After reviewing the documentation, assign a score from 1 to 3 for each of the following criteria:
- Completeness: measures the percentage of crucial information present.
- Clarity: evaluates the comprehensibility of the documentation.
- Consistency: assesses the coherence of information in the documentation.
- Accessibility: assesses how easily users can access and read the documentation.

2.2.5 Human evaluation

Human evaluation remains a fundamental step in the evaluation process. Especially for our use case, where automatic evaluations are not immediately applicable and the textual document presents several layers of complexity, this type of assessment offers a reliable and straightforward way to evaluate the output quality. Generally, this metric is more expensive in terms of both time and resources used. In fact, human evaluation will be carried out by involving expert developers in the considered

programming language, who will analyze the output and assess it according to the criteria and scores provided by us. Such criteria are the same utilized for the LLM evaluation. In Table 2.3, we present these metrics and their corresponding scores, as presented to the human annotator before the evaluation.

Criterion	**Description**	**Score Scale**
Completeness	Measures the percentage of crucial information present.	1-3
Clarity	Evaluates the comprehensibility of the documentation.	1-3
Consistency	Assesses the coherence of information in the documentation.	1-3
Accessibility	Assesses how easily users can access the documentation.	1-3

Table 2.3: Evaluation Criteria and Scores for Human Evaluation

2.2.6 Additional Metrics

Additional metrics to consider are the time taken to produce the output and the associated costs. The evaluation criterion follows a "less is better" approach, but this is balanced by other metrics: a free but inaccurate output is likely to be rated lower than a comprehensive one that incurs a cost. A comprehensive assessment for time and costs is sumarized in Table 2.4.

Metric	**Ideal**	**Good**	**Bad**
Time	<1 min	1-10 min	>10 min
Cost	Free	<$5	>$5

Table 2.4: Time and Cost assessment

2.3 Final Comparison

Once all the scores have been obtained, the metrics from the evaluation framework will be summarized in two final tables.

The first table will encapsulate the scores achieved by the various approaches in both the LLM evaluation and the human evaluation, alongside the time taken to generate the output and the associated costs. The scores obtained for each evaluation criterion will be summed and presented as an overall score on a 12-point scale. For example, a documentation that received 2/3 in each of the four criteria will have a final score of 8/12 reported in the table. The best results and the the ones considered "Good" or "Ideal" based on our assessment tables, will be highlighted in bold. This table provides an immediate and comprehensive view of which approaches performed best, allowing us to observe differences and potential correlations between the different types of evaluation.

The second table will present the scores for the automated metrics (ROUGE scores and BERTScore) for each approach. This table will be crucial for identifying any differences between open-source and closed-source approaches in understanding and summarizing the content of various programs, as well as for assessing the performance of the metrics themselves. When the metrics are calculated based on multiple documents, an arithmetic average is reported. Again, scores considered "Good" or "Ideal" based on our assessment tables, will be highlighted in bold.

Chapter 3

Prompt engineering

3.1 Definition

"Prompt engineering is the process of writing, refining and optimizing inputs to encourage generative AI systems to create specific, high-quality outputs." [3] It consists of crafting specific natural language inputs, known as prompts, to guide the model's output towards desired results. A prompt can be defined as a set of instructions provided to an LLM that programs the model by customizing it and enhancing or refining its capabilities [32].

This chapter explores some of the most common methodologies, techniques, and best practices of prompt engineering, illustrating how it can transform the interaction with LLMs into a programmable and customizable experience and how it can enhance their performance for our use case.

3.2 Zero-Shot prompting

Zero-shot learning involves providing the LLM with a prompt that does not include any examples or prior context related to the task. The model relies only on its pre-trained knowledge to generate a response. This approach tests the model's ability to generalize from its training data to new, unseen scenarios. For instance, a zero-shot

[3] IBM definition of prompt engineering [18]

prompt might ask the model to classify the sentiment of a sentence.

Prompt

Classify the text into neutral, negative or positive.

Text: I think the restaurant is okay.

Sentiment:

Output

Neutral

The prompt above does not provide the model with any examples of text alongside their classifications, the LLM already understands "sentiment" thanks to its pre-trained knowledge.

Instruction tuning has been shown to improve zero-shot learning [32]. With instruction tuning models are fine-tuned on datasets described via instructions: the model's parameters are adjusted to better align with the requirements of these tasks such as question answering, without losing the general language understanding it has already acquired.

However, especially for complex or not straight-forward tasks, it is always recommended to provide demonstrations or examples in the prompt, which leads to few-shot prompting.

3.3 Few-Shot prompting

Few-shot prompting, which enables in-context learning, provides the LLM with a few examples within the prompt to guide its response. This method leverages the model's ability to learn from the context provided by these examples to perform the task more accurately. In the following example, presented in Brown et al. 2020 [31], the task is to correctly use a new word in a sentence.

Prompt

A "whatpu" is a small, furry animal native to Tanzania. An example of a sentence that uses the word whatpu is:
We were traveling in Africa and we saw these very cute whatpus.

To do a "farduddle" means to jump up and down really fast. An example of a sentence that uses the word farduddle is:

Output

When we won the game, we all started to farduddle in celebration.

The model has learned how to perform the task by providing it with just one example (i.e., 1-shot). For more difficult tasks, more demonstrations may be provided. Standard few-shot prompting works well for many requests and can be easily combined with other methods to help delivering the best possible output. However, it is still not a all-around technique, especially when dealing with more complex reasoning tasks. More recently, chain-of-thought (CoT) prompting has gained popularity to address more complex arithmetic, commonsense, and reasoning assignments.

3.4 Chain of Thought

The chain of thought methodology involves structuring the prompt in a way that encourages the LLM to reason through a problem step-by-step. This technique is particularly useful for complex tasks that require logical reasoning or multi-step calculations. By breaking down the task into smaller, manageable steps, the model can follow a clear logical progression to arrive at the solution. For example, solving a math problem using a chain of thought prompt would involve guiding the model through each step of the calculation process.

Prompt

Q: Riccardo has 52 cards. He buys 2 more deck of cards. Each deck contains 52 cards. How many cards does he have now?

A: Riccardo started with 52 cards. 2 decks of 52 cards each are 104 cards. 104 + 52 = 156. The answer is 156.

Q: The supermarket had 11 employees. If they fire 5 of them and hire 3 more, how many employees do they have?

Output

The supermarket had 11 employees originally. They fire 5 of them. So 11 - 5 = 6. They hire 3 more employees, so they have 6 + 3 = 9. The answer is 9.

Furthermore, it is possible to add more examples, combining CoT with few-shot prompting to get better results and increase the accuracy on complex tasks that require reasoning before responding.

A recent idea that has emerged is the concept of zero-shot CoT (Kojima et al. 2022 [22]), which essentially involves adding the sentence "Let's think step by step" to the original prompt.

Prompt

I went to the market and bought 10 apples. I gave 2 apples to the neighbor and 2 to the repairman. I then went and bought 5 more apples and ate 1. How many apples did I remain with?
Let's think step by step.

Output

First, you started with 10 apples. You gave away 2 apples to the neighbor and 2 to the repairman, so you had 6 apples left. Then you bought 5 more apples, so now you had 11 apples. Finally, you ate 1 apple, so you would remain with 10 apples.

The presence of the simple prompt "Let's think step by step" has proven to lead to great effectiveness in this kind of tasks. This is particularly useful when there are not too many examples to use in the prompt.

New LLMs are progressively getting better in answering to such assignments and CoT is not always necessary, but it is still the best practice to obtain more solid results.

3.5 ReAct

The paper *React: Synergizing reasoning and acting in language models* [46] introduced a framework named ReAct where LLMs are used to generate both reasoning traces and task-specific actions in an interleaved manner. This prompting methodology is designed to not only make the model think through a problem, but also take specific actions based on its reasoning. The ReAct framework can also allow LLMs to interact with external tools to retrieve additional information that leads to more reliable and factual responses. The ReAct prompting technique, illustrated in Figure 3.1, seeks to enhance the model's performance combining reasoning and acting by integrating both cognitive and action-oriented components within the prompt. In the Reason Only paradigm, the LLM generates reasoning traces to support its output. Here, the reasoning is done upfront, which helps the model perform better since the thought process precedes the final answer and it is not a mere justification of the output, as often happens. The Act Only paradigm involves the LLM inter-

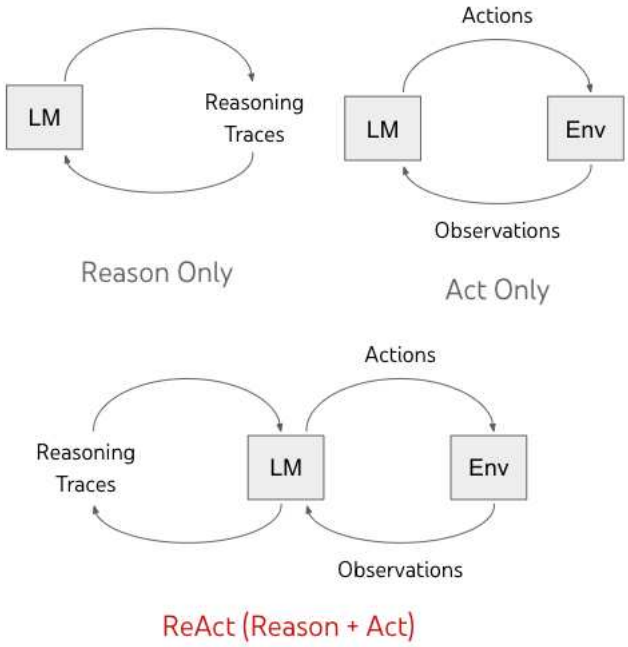

Figure 3.1: ReAct - Reason + Act paradigm [45]

acting with an environment. The model performs an action in this environment, receives feedback as observations, and uses these observations to refine subsequent actions. This approach helps the model learn and adapt based on real-world interactions, enhancing its ability to perform tasks that require dynamic responses. ReAct combines these two paradigms to achieve superior results. It starts with reasoning, allowing the LLM to form an initial understanding or hypothesis. Then, the action step allows to interface with and gather information from external sources such as knowledge bases (e.g. Wikipedia) or environments. So, the model takes an action based on this reasoning and receives observations from the environment or external source. These observations are fed back into the LLM, enabling it to refine its reasoning in a multi-step process. Unlike chain-of-thought (CoT) prompting, which typically involves a single sequence of reasoning steps, ReAct employs a multi-shot approach where reasoning and actions iteratively improve the model's performance. This iterative cycle of reasoning and action allows the LLM to handle complex tasks more effectively. The integration of both paradigms ensures that the model not only

thinks through problems thoroughly but also learns from its interactions with the environment, leading to more accurate and reliable outputs.

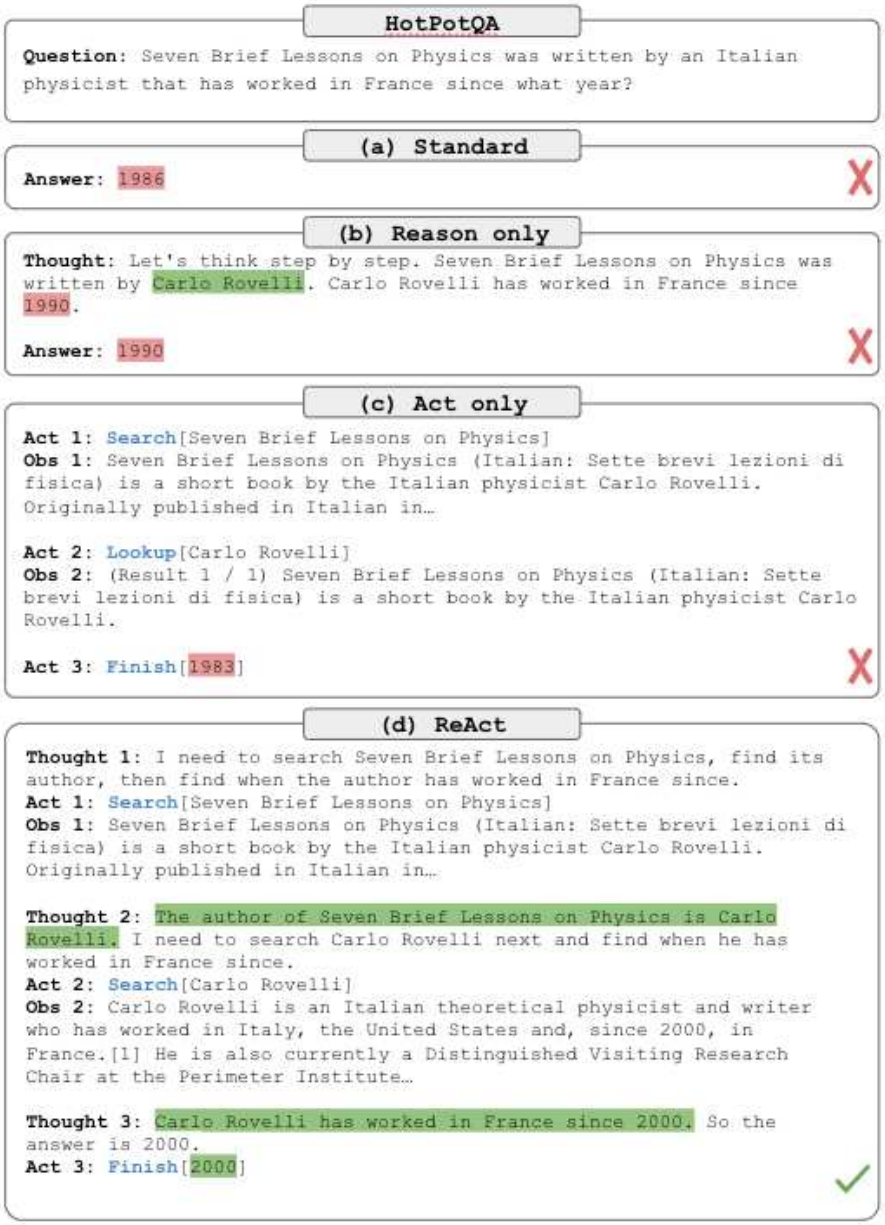

Figure 3.2: ReAct - prompting example [46]

A practical example of ReAct prompting technique can be further elucidated with the example in Fig. 3.2, presented in the paper [46] and showcasing the process of answering a question about the book *Seven Brief Lessons on Physics*. The com-

parison between different prompting methods—Standard, Reason Only, Act Only, and ReAct—highlights the distinct advantages of ReAct. In the Standard method (a), the model directly provides an answer, often incorrect, without any reasoning or action steps. The Reason Only approach (b) incorporates some chain-of-thought reasoning, which gives a bit of context but still results in an incorrect answer. The Act Only method (c) involves searching and retrieving information, but it lacks the iterative reasoning necessary to refine the results, leading again to an incorrect answer. The ReAct prompting technique (d) integrates both reasoning and action in a multi-step process:

1. **Initial Thought**: The model starts with a thought, initiating the reasoning trace. In this example, the thought is to find the author of *Seven Brief Lessons on Physics*.

2. **First Action**: The model performs a search based on this thought, retrieving information about the book and its author, Carlo Rovelli.

3. **Observation**: The retrieved information is used as an observation, feeding back into the model.

4. **Second Thought**: The model then reasons that it needs to find when Carlo Rovelli started working in France.

5. **Second Action**: Another search is conducted, retrieving information about Rovelli's career.

6. **Final Observation**: The model observes that Rovelli has worked in France since 2000.

7. **Final Thought and Answer**: Using this observation, the model concludes with the correct answer: 2000.

By combining reasoning with actionable steps and incorporating feedback at each stage, ReAct can handle complex questions more effectively than standard prompting methods. Although ReAct uses more tokens and requires multiple calls to the

LLM compared to standard prompting, the results outperform several state-of-the-art baselines especially on language and decision-making tasks. The multi-step reasoning and action process ensures that each piece of information is validated and contextualized, leading to more accurate, truthworthy and reliable answers.

3.6 Role prompting

Role prompting, also known as persona-based prompting, is a technique used to guide a model's responses by assigning it a specific role or persona. This method involves starting the prompt with phrases like "As a teacher...", "As an expert in..." or "You are an expert in..." to frame the context and expected style of the response. By doing so, the model can tailor its output to align with the characteristics and expertise associated with the given role. This approach is particularly useful for generating context-specific information, maintaining consistent tone and style, and enhancing the relevance and clarity of the responses. For instance, a persona-based prompt can make the model avoid sentences like "Let me know if I can do something else for you" or "Here's the text you requested", making the output cleaner and neutral.

Prompt

As a front-end developer, review and eventually fix the following code:

```
<code> {{CODE}} </code>
```

Further experiments on role prompting have led to the development of derived techniques such as role-play prompting, proposed in the paper *Better Zero-Shot Reasoning with Role-Play Prompting* [23].

3.7 Prompt-Chaining

To enhance the reliability and performance of large language models, another effective prompt engineering technique is breaking down tasks into subtasks. After identifying these subtasks, the LLM is prompted with each of them sequentially, using the response from one as the input for the next. This method, known as prompt chaining, involves creating a series of prompt operations to tackle complex tasks that an LLM might find challenging if given just a highly detailed prompt. In this approach, each prompt applies transformations or additional processes to the generated output before reaching the final outcome. Beyond improving performance, prompt chaining enhances the transparency, controllability, and reliability of LLM output. This has proven to allow for easier debugging of model responses and it facilitates performance analysis and improvements at each stage of the task. Concretely, prompt chaining is valuable for developing LLM-powered conversational agents and assistants, and for enhancing personalization and user experience in applications. For example, when using an LLM to answer questions about a large text document or code, two distinct prompts can be designed. The first prompt extracts relevant quotes to answer the question, and the second prompt uses these quotes and the original document to generate a final answer. This method effectively utilizes two separate prompts to accomplish the task of answering a question based on the document. Here's some other use cases reported in Anthropic documentation [5] for Claude models, which suit well for long text analysis thanks to their large context window:

- Multi-step analysis: *Analyze → Write suggestions → Refine.*
- Content creation pipelines: *Research → Outline → Draft → Edit → Format*
- Data processing: *Extract → Transform → Analyze → Visualize*
- Decision-making: *Gather info → List options → Analyze each → Recommend*
- Verification loops: *Generate content → Review → Refine → Re-review*

The following is an example of self-correcting summary with multi-prompt approach provided in the Claude documentation:

Prompt 1

Summarize this medical research paper.

```
<paper> {{RESEARCH_PAPER}} </paper>
```

Focus on methodology, findings, and clinical implications.

Prompt 2

Your task is to provide feedback on a research paper summary. Here is a summary of a medical research paper:

```
<summary>
{{SUMMARY}}
</summary>
```

Here is the research paper:

```
<paper>
{{RESEARCH_PAPER}}
</paper>
```

Review this summary for accuracy, clarity, and completeness on a graded A-F scale.

Prompt 3

Your task is to improve a paper summary given feedback. Here is the first draft of a medical research paper:

```
<summary>
{{SUMMARY}}
</summary>
```

Here is the research paper:

```
<paper>
{{RESEARCH_PAPER}}
</paper>
```

Here is the feedback:

```
<feedback>
{{FEEDBACK}}
</feedback>
```

Update the summary based on the feedback.

As we can see in the examples, it is standard to put documents and outputs from previous calls inside HTML tags in the prompt.
Prompt-chaining has being fundamental to develop our multi-agent approach (Section 4.5) to produce clean and refined code documentation.

3.8 Automatic Prompt Engineer (APE)

In the last two years, several methods for automatically optimizing prompts have been proposed, exploiting iterative and search algorithms. In the paper Zhou et al., (2022) [50], the authors propose a framework for automatic instruction generation and selection: automatic prompt engineer (APE). Through the use of a inference

LLM that is given input-output pairs demonstrations, prompt candidates are generated for a target task. These candidate solutions are executed using another model and then the most accurate prompt is selected based on computed evaluation scores.

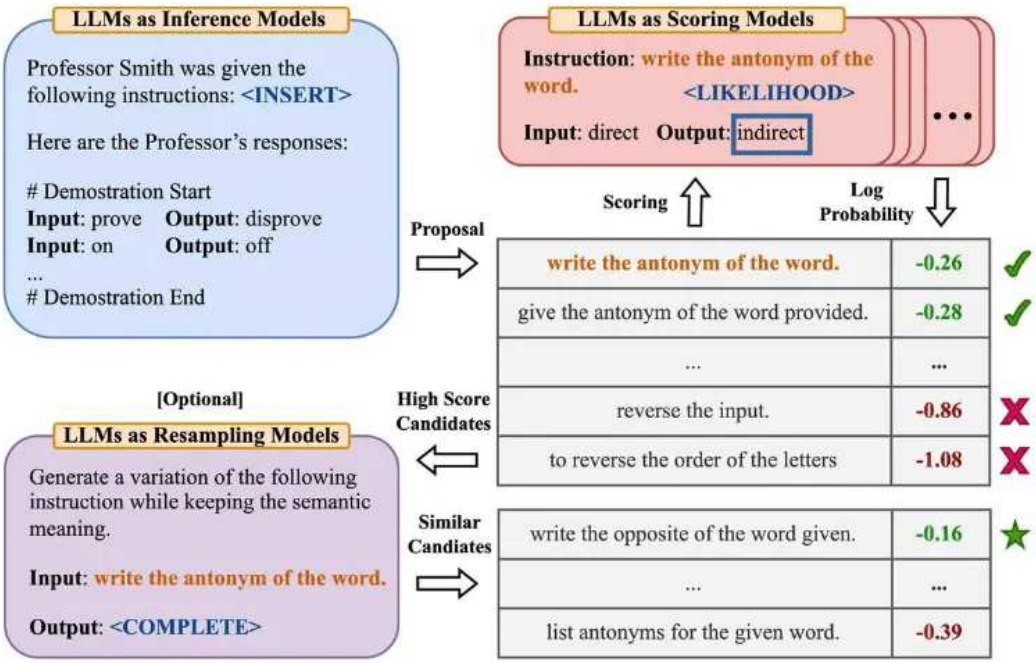

Figure 3.3: APE schema, Zhou et al. 2023 [50]

Automatic prompt generation and optimization is a newsworthy topic that will certainly play an increasingly significant role in the field of generative AI.

3.9 Prompts for the use case

It is not possible to reduce the prompt engineering work to a single prompt; however, we can outline general characteristics used during the implementation of individual approaches. Further detailed considerations can be found in each specific section. As mentioned several times, each model responds differently to the same prompt, and this can also vary based on the model's capabilities. It is always advisable to refer to the documentation of the language model used for the most suitable prompting techniques. Anthropic provides essential tips for long-context prompts, which have

been very useful given our extensive use of Claude models. Below, we present some of these instructions that we have used:

- Put longform data at the top
- Structure document content and metadata with XML tags

Placing long inputs (20K+ tokens) near the top of the prompt, above queries, instructions and examples can significantly improve Claude models' performance. The importance of XML or HTML tags has already been highlighted in previous sections, but it is still formalized: wrapping each element, such as documents, in *<document>* tags and other metadata subtags is essential to enhance LLMs' performance. Together with these considerations, techniques such as role prompting, zero and few-shot prompting and prompt chaining have been standard in all approaches, while prompt-chaining have been used for the multi-agent approach (section 4.5). For some experiments, the same LLMs were used to generate possible prompts based on the task description and a reference output. While this cannot be defined as APE, it certainly resembles that approach. Below, we provide an example of the structure of a generic instruction given for creating code documentation.

Prompt Example

You are a {language} expert, looking at the {language} code program {program_name}, your task is to perform a reverse engineering process to write a short functional specification document. Use the in-line code comments to help understand the context, if they are presents. The {language} program:

```
<code>
{{CODE}}
</code>
```

The document must contain the following sections:

```
< list and description of each section >
< additional context >
```

The output must be structured, detailed but as concise as possible. Avoid including design elements or technical information. The document must be written in plain English and use markdown format. Ensure clarity and readability to aid understanding. Use a clear hierarchy of headings and subheadings to organize the content. Include a clickable table of contents at the top with hyperlinks to each section. Include relevant pseudocode snippets to illustrate each section. Embed all diagrams using Mermaid syntax.

As expected, the prompt is customized based on the type of input program, adjusting variables related to the language, program name, and included sections. Moreover, an additional context is added, depending on the type of code under analysis. Some examples of context can be:

Additional Context - Example 1

As additional context, here's a brief description of the Euronext Securities Milan (ESM) and its information system (SIMT aka Sistema informativo Monte Titoli). The following details are only to give you some more context about the program and they should not reported as narrative text. Use them only to better understand the code scope and to the process of resolving conflicts that arise when a concept can have different meanings.

Additional Context - Example 2

As additional context, here's a subset of the Euronext Securities Milan's DB2 database tables names with their descriptions. Please refer to any of these tables only if it's present in the {language} code program {program}. Do not include in your answer any of them if they aren't explicitly present or referred in the {language} code program {program_name}. The following details are only here to give you some more context about the program and they should not reported as narrative text. Use them only to better understand the code scope and to the process of resolving conflicts that arise when a potential concept can have different meanings. Only if you discover a table name presents in the program, you can try to decode its description using this list:

More specifications about the output format can be done, especially when the model seems to have difficulties producing objects such as diagrams or when there are errors in the general formatting.

Prompt for Output Format Specifications

Output Format:

- The documentation should be written in GitHub Flavored Markdown (GFM) format
- Use # for top-level headings, ## for second-level headings, and so on
- Ensure all code snippets are fenced with triple backticks (“‘) and include a language specifier for syntax highlighting (e.g. “‘cobol)
- Use **bold** for emphasis and ‘inline code‘ for short code references
- Include clickable links to external references or other sections within the document
- Embed all diagrams in the markdown using Mermaid.js syntax (see below)
- Provide a clickable table of contents at the top of the document with links to each section

Diagram Specifications:

- All diagrams should be created using Mermaid.js syntax and embedded directly in the markdown
- Use the following Mermaid diagram types for each section:
  1. Overview: Flowchart Diagram
  2. Data Structures: Entity-Relationship Diagram
  3. Program Flow: Flowchart Diagram
  4. Error Handling and Logging: Sequence Diagram
  5. External Interactions: Component Diagram
  6. Database Operations: Sequence Diagram
  7. Transactional Behavior: State Diagram
- Each diagram should be preceded by a Mermaid code block wrapped in mermaid tags
- Use meaningful names and labels for each diagram element
- Include a brief caption below each diagram explaining its purpose
- Ensure all text is readable and the layout is clear and uncluttered

Here is an example of a Mermaid flowchart:

Provide examples (few-shot) of the right formatting of the Mermaid diagrams have proven to be effective in improving their quality, expecially in less powerful models.

Chapter 4

Experiments and results

In this chapter, we present the various approaches we tested for the use case of producing automatic code documentation using generative AI. The goal was to identify efficient, cost-effective methods that can be seamlessly integrated into the existing workflows. To achieve this, firstly, we engaged with three prominent consulting firms, each of which presented their proprietary solutions utilizing closed-source models. These consultations provided us with valuable insights into state-of-the-art commercial offerings and their capabilities in the landscape of automatic code documentation and mainframe migration. Simultaneously, we explored open-source solutions with the aim of achieving comparable results at a lower cost and with greater internal control. The open-source approaches were carefully tested and are presented in detail within this chapter, together with their counterpart. By examining and comparing both closed-source and open-source options, we aimed to balance performance, cost, and flexibility, ultimately striving to find the most suitable solution for our needs.

4.1 Summary of the Approaches

This section provides a structured overview of the different approaches explored in our experiments. These approaches represent a spectrum of techniques and tech-

nologies, from commercial closed-source solutions provided by external consultants to innovative open-source methods. Each approach was evaluated for its performance, cost, and suitability for integration into the existing system. Below is a brief summary of each approach, with a detailed analysis to follow in subsequent sections.

- **External Provider Approaches:**

  Involve consulting firms with their own solutions and POCs, offering an overview of powerful commercial tools but with the trade-offs in costs and data privacy.

- **Direct Approach:**

  This method uses a single LLM to generate documentation directly from the code, providing a straightforward and efficient solution. The direct approach will be analyzed for both closed-source and open-source models.

- **Chain Approach:**

  This method involves the usage of Chains to overcome context window limitation and it is particularly useful with open-source LLMs.

- **Multi-agent Approach:**

  A collaborative method where multiple specialized LLMs work together to produce comprehensive and refined documentation. The multi-agent approach will be carried out and evaluated both for closed-source and open-source models.

In the sections that follow, we will provide an analysis of each solution proposed by the consulting firms, followed by a comprehensive exploration of our internal experimentations, both with closed-source models and open-source alternatives. This comparative study will highlight the strengths and weaknesses of each approach, offering a clear perspective on the most effective strategies for leveraging LLMs in the automatic production of code documentation.

4.2 External providers solutions

As previously mentioned, the feasibility study for the task of producing code documentation and reverse engineering mainframe applications was initially conducted on two fronts: internally and through expert consulting firms. The consulting process, carried out through proofs of concept (POC) presented by distinct companies, demonstrated the success of the task and was useful for uncovering the potential of the most powerful models on the market and tools designed by external providers, drawing ideas and inspiration for alternative solutions. Through an iterative feedback process over several weeks, typical of POCs, these companies were able to produce satisfactory documentation for the provided sample programs. In the following sections, we will analyze the results, extracting useful conclusions and insights that will be utilized for internal development.
For privacy reasons, Euronext Securities' providers are anonymized.

4.2.1 Company A

Company A proposed a solution leveraging Anthropic's Claude 3 Opus model to automatically generate functional documentation for the sample programs (1.3). Claude 3 Opus is renowned to be the most powerful Anthropic's model, only recently overcame by Claude 3.5 Sonnet, with a context window of 200k tokens, making it particularly well-suited for tasks involving long codebases and document. Upon receiving our request, Company A delivered two sets of functional documents, one for COBOL and the other for EGL. These initial documents were impressively detailed, showcasing the model's ability to understand and articulate the intricacies of the programs. We then requested some refinements to better align the documentation with our specific requirements. Company A adeptly adapted the outputs, resulting in final documentation that was both comprehensive and precise, meeting our standards for quality and accuracy. This POC was very instructive for the subsequent internal development.

4.2.2 Company B

Company B presented a more complex solution, using GPT-4o to produce documentation and a reverse engineering tool for modernizing the development of IBM mainframe COBOL and PL/1 applications: MicroFocus Enterprise Analyzer. The tool allows managing complex application portfolios and accelerates modernization projects, eliminating low-quality and redundant code. Analyzer also helps isolate logic and re-architect systems. The documentation obtained, after several adjustments, was satisfactory. However, integrating the external tool proved to be complex and challenging to demonstrate in a POC.

4.2.3 Company C

Company C's solution, though not directly tested on ESM data, presented a promising approach to reverse engineering legacy applications using GPT-4-Turbo and Retrieval-Augmented Generation (RAG) see Section 5.2. The application divides the code and leverages RAG to overcome context window limitations, enabling processing of large codebases. It initializes the LLM engine and RAG components, then loads and preprocesses source code files, embedding code snippets into a vector database. For each code structure, it generates initial documentation using the LLM, employs RAG to retrieve relevant context, and enhances the documentation accordingly. An LLM agent then validates and refines this documentation. The process concludes by combining all documentation chunks, performing final validation and consistency checks, and outputting comprehensive documentation in markdown format. Although still in development, this solution impressed us with its structured approach, encouraging similar methods and providing valuable insights for internal testing. The use of RAG and agents will be explored further in subsequent sections.

4.3 Direct Approach with Large Context

The nature of the documents to be analyzed, especially those written in COBOL, posed a significant challenge for creating the corresponding documentation. The WSOC83 program has a base length of approximately 60k tokens, exceeding the context window of many models, which are therefore unable to process it correctly. Moreover, Euronext's mainframe system contains applications with programs even longer than WSOC83, necessitating a scalable solution that could be applied to any code received as input, not just the example code used for experiments. Needless to say, the best and easiest way to achieve this result is to use a direct approach with an LLM that has a large context window. On the closed-source front, Claude 3 Opus and 3.5 Sonnet, and GPT-4o are perfect as they are designed to handle a large amount of text input (Fig. 1.3). Additionally, being state-of-the-art in the LLM world, they achieve very high benchmark results on code. Moreover, Anthropic states that Sonnet 3.5 "handles code translations with ease, making it particularly effective for updating legacy applications and migrating codebases." [5]. Regarding open-source models, as reported in Section 1.5.3 and shown in Fig. 1.5, few have a sufficiently large context window (and still smaller than proprietary LLMs) in their base version. However, strategies and measures can be implemented to ensure scalability even in the open-source context.

4.3.1 Application

The application for the Direct Approach with Large Context is implemented in Python, utilizing tools described in Section 1.4. It's designed to invoke desired models either locally or via APIs, and manages the prompting process through a mechanism detailed in the next section. The application reads the target programs (COBOL, PL/1, or EGL) from a local directory and processes them. To overcome potential limitations of open-source models, a custom "cleaner" is applied for each programming language. This cleaner removes unnecessary spaces, effectively re-

ducing the total number of tokens while preserving the program's meaning. For example, the WSOC83 program is reduced from 60k tokens to 43k tokens through this process. For experimental purposes, the application allows for the translation or removal of inline comments, enabling observation of how models behave under various scenarios. By default, comments are translated and retained. Finally, the chosen model is invoked with the correct prompt. While many LLMs were experimented with, for this specific direct approach, the application allows selection between CodeQwen1.5 7B-Chat (run locally) and Claude 3-Opus (called through a pay-per-use API).

4.3.2 Prompting

The process of prompt engineering required the most time and attention in the development of this application. The prompt used varies based on the type of documentation desired and the model chosen. The application implements both single-chapter and multi-chapter approaches to documentation generation. In the single-chapter approach, the entire documentation is generated in one go, while the multi-chapter approach breaks down the documentation into separate sections, each generated independently. Context is added to the prompts to provide additional information about the specific environment and requirements of Euronext Securities Milan. This context helps guide the model in generating more relevant and accurate documentation.

4.3.3 Evaluation

For the evaluation, we will primarily focus on the COBOL functional documentation, which is the main objective of our experimentation. We will also comment on the results obtained for the other languages, highlighting any differences and trade-offs, if present.

4.3.3.1 Closed-source results

- **OQE**

  The final documentation produced using the Direct Approach with the closed-source model Claude 3.5 Sonnet is both satisfactory and comprehensive, featuring a high degree of detail and accuracy. The output is well-formatted, appears to cover all the required sections and even includes charts and diagrams to complete some of the descriptions.

- **Benchmark Evaluation**

  Comparing the AI-generated documentation with pre-existing human-written versions reveals noticeable differences in style and topic organization. These variations are likely due to the absence of a predefined structure in the human-written documentation, which contrasts with the consistency provided by structured and uniform prompts. Beyond these formatting differences, the automatically generated documentation appears to be more direct and concise, as well as more fluid and readable. It provides a high level of detail about the program's functionality, including graphical elements that are missing in the benchmark version. However, where it falls short is in contextualizing the program within the company's systems. The benchmark documentation contains much more specific information about Euronext and its systems, with references to details that are impossible to catch from the code alone. The only way to bridge this knowledge gap is to provide the model with additional, specific context in the prompt, beyond what is already included. This is a challenging task to generalize for every potentially analyzable program.

- **Evaluation with LLM**

  The LLM evaluation is run on the UTIP0060 COBOL program. The output and the scores of the automated LLM evaluation are the following:

  1. **Completeness:** 3/3

     The documentation is comprehensive, covering all crucial aspects of the

program. It includes detailed information about the program's purpose, data structures, main functionalities, process flow, error handling, and even test cases. The addition of subroutines, unexpected cases, and testing guidelines greatly enhances the completeness of the documentation.

2. **Clarity:** 3/3

   The documentation is very clear and well-organized. It uses plain language to explain complex concepts and provides visual aids like flowcharts to illustrate the program flow. The breakdown of features, variables, and data structures is presented in a logical and easy-to-understand manner. The addition of specific error messages and their meanings also contributes to the clarity.

3. **Consistency:** 3/3

   The information presented in the documentation is highly consistent throughout. The program flow described in the text matches the provided flowchart. The descriptions of data structures, variables, and functionalities are consistent with the actual COBOL code. There are no apparent contradictions or inconsistencies in the information provided.

4. **Accessibility:** 3/3

   The documentation is extremely accessible. It's well-structured with clear headings and subheadings, making it easy to navigate. The use of flowcharts and bullet points enhances readability. The language used is clear and appropriate for technical documentation, and complex concepts are broken down into manageable sections. The inclusion of a testing guide also makes it more accessible for different types of users, including developers and testers.

In summary, the documentation is of excellent quality. It provides a comprehensive, clear, consistent, and accessible description of the UTIP0060 program. The documentation covers all aspects of the program, from its purpose and

data structures to error handling and testing guidelines. This level of detail and organization greatly enhances the usability and maintainability of the program.

- **Human Evaluation**

  The human evaluation was carried out by myself and two expert developers in COBOL, who analyzed the output and assessed it according to the criteria and scores provided by us and reported in Section 2.2.5. The average scores are reported in the following table:

Metric	**Score**
Completeness	2.7/3
Clarity	2.7/3
Consistency	2.7/3
Accessibility	2.3/3

Table 4.1: Human Evaluation - Direct Approach with Closed-source model

The results, presented in Table 4.1, are consistent with what we obtained in the LLM evaluation. The generated documentation has an overall excellent quality, it is comprehensive and consistent. However, there is room for improvement in terms of accessibility and formatting which can be obtained through a strighter prompting.

4.3.3.2 Open-source Results

The outputs we decide to evaluate are produced using the model CodeQwen1.5-chat. Similar results have been obtained with CodeGeeX4, while other models achieved less satisfactory outputs.

- **OQE**

  The documentation produced using the Direct Approach with the open-source

models looks well-formatted, complete, with a good level of detail. All the required sections appear to be featured, together with diagrams. Overall, the document seems to be satisfactory.

- **Automated Metrics**

  The automated metrics are computed on the closed-source benchmark documentation. Again, the programs "SYSPTAB5" and "UTIP0060" has been taken as COBOL evaluation example.

Metric	SYSPTAB5	UTIP0060
ROUGE - 1	0.42	0.38
ROUGE - 2	0.05	0.14
ROUGE - L	0.21	0.32
BERTScore	0.62	0.62

Table 4.2: Automated Metrics on "SYSPTAB5" and "UTIP0060" programs, computed against closed-source benchmark - Direct Approach

  The automated metrics reveal that the documentation produced by the open-source model, in its general overview of the program, successfully captures and describes the program's functionality and structure, similar to what the commercial model achieved. The ROUGE scores indicate good results in terms of overlapping single words used, though they are less promising when considering longer sequences of terms. However, the BERTScore is quite high, suggesting that the overall meaning of the sentences is comparable to the reference, despite some potential differences or gaps in the terminology used.

- **Evaluation with LLM**

  The LLM evaluation is run on the UTIP0060 COBOL program. The output and the scores of the automated LLM evaluation are the following:

  1. **Completeness:** 2/3

The documentation covers most of the crucial information, including the program's purpose, main functionalities, data structures, program flow, and some error handling. However, it lacks detailed information about specific procedures and subroutines within the program. Additionally, while it mentions database operations, it doesn't provide specific details about the tables or SQL queries used.

2. **Clarity:** 2/3

   The documentation is generally clear and understandable. It uses diagrams (flowcharts, entity-relationship diagrams, sequence diagrams) to illustrate program flow and data structures, which enhances clarity. However, some sections, particularly those describing the algorithms and techniques used, could benefit from more detailed explanations or examples.

3. **Consistency:** 2/3

   The information presented in the documentation is mostly consistent. The program flow described in the text matches the flowchart provided. However, there are some inconsistencies between the documentation and the actual COBOL code. For example, the documentation mentions database operations and external interactions that are not evident in the provided COBOL code.

4. **Accessibility:** 3/3

   The documentation is well-structured with clear headings and subheadings, making it easy to navigate. The use of various diagrams (flowcharts, entity-relationship diagrams, sequence diagrams) enhances the accessibility of the information. The language used is generally clear and appropriate for technical documentation.

In summary, while the documentation provides a good overview of the program and uses helpful visual aids, there's room for improvement in terms of completeness and consistency with the actual code. More detailed explanations

of specific procedures and accurate representation of the program's actual functionalities would enhance its overall quality.

- **Human Evaluation**

  The human evaluation was carried out by myself and two expert developers in COBOL, who analyzed the output and assessed it according to the criteria and scores provided by us and reported in Section 2.2.5. The average scores are provided in the following table.

Metric	**Score**
Completeness	2.3/3
Clarity	2/3
Consistency	3/3
Accessibility	2/3

Table 4.3: Human Evaluation - Direct Approach with Open-source model

The results, presented in Table 4.3, are quite similar to what we had in the LLM evaluation. Overall, the generated documentation is satisfactory, but it could definitely be improved. The level of detail about certain operations could be increased, adding more flows, diagrams and example. The documentation is well structured, but it could include more links to the different sections to make the navigation easier.

4.3.3.3 PL/1 and EGL results

As mentioned earlier, the primary goal of the experiment is to evaluate how our solution performs on COBOL programs. However, we decided to expand our scope and conduct some tests on PL/1 and EGL programs as well, which have already been extensively described in the specific section. We will now provide a more qualitative and less technical commentary on the results obtained with these additional programs.

The application's performance in generating documentation for PL/1 and EGL programs is comparable with what we obtained with COBOL, demonstrating promising capabilities in capturing each program elements. It successfully extracts and organizes key information such as functional overviews, technical details, program flow, and change history. Thus, the generated documentation provides a solid foundation for understanding the program's purpose and structure. However, as often seen with COBOL, there is room for improvement in areas such as in-depth explanations of complex processes, visual representations of data flow, and coverage of error handling mechanisms. Future enhancements to the application could focus on specific prompts for EGL and PL/1, such as the ones already applied (like the translation of in-line comments from Norwegian or Dutch to English), to produce more comprehensive and visually intuitive documentation. Despite these areas for improvement, the output obtained is a valuable starting point for developers and maintainers working with legacy PL/1 and EGL systems in Euronext Oslo and Copenhagen, significantly reducing the time and effort required to understand and maintain such programs.

4.3.3.4 Final Considerations

The application using the direct approach reaches excellent results and appears to be a viable alternative for our experimentation. The documentation produced with Claude 3.5 Sonnet is of amazing quality, detailed, and complete, with costs ranging from $2 to $5 per document. Although the open-source models provide slightly less precise results and fall short in completeness compared to the actual code, they have the significant advantage of being free, along with other characteristics that have been extensively discussed. The average time taken to generate the functional documentation is less than a minute for Claude and 5 to 10 minutes for CodeQwen, based on the program's length. As the size of the COBOL program increases, the local model begins to struggle to generate coherent and relevant documentation. The models with the smallest CWs are not able to take some of the programs as input or their outputs result to be cut, due to the maximum number of tokens reached.

This is where commercial LLMs, with larger context windows and more parameters, can clearly offer superior results. The application also supports manual review of the generated documentation, allowing further refinement based on feedback from domain experts.

As an additional test, we decided to remove the inline comments from the code and assess how the models performed on such a sample. The programs used in the experiments had numerous comments (as they should) explaining their functionality, which undoubtedly helps LLMs understand and describe the code. To assess the actual influence of these comments, we removed them and conducted further tests. The resulting documentation was significantly impacted by the absence of comments, showing a lower level of detail and completeness. While the outcome can still be considered satisfactory, more complex and uncommented programs could lead to a drastic decline in quality. This issue may depend on the number of applications and programs within Euronext's system that were not properly commented during creation or revision, making it a factor to consider for the potential adoption of the AI-based solution.

4.4 Overcome CW limitations with Chains

When we work with large documents, we can face some challenges as the input text might not fit into the model context length, the model hallucinates with large documents or run into out of memory errors. In these cases the direct approach with a large context window is not feasible. To solve those problems, we are going to show a solution that is based on the concept of chunking and chaining prompts. We decided to introduce in our application summarization methods called *chains* to be able to work with a greater variety of models and to ensure additional robustness even when processing extremely long input programs.

4.4.1 Chains Integration

A common summarization method is the MapReduce chain provided by LangChain. This approach divides the document into smaller, manageable chunks that fit within the model's context window. Each chunk is processed independently by the language model, which generates partial results. These partial results are then aggregated to form an output. This technique effectively extends the model's capacity to handle larger documents by breaking down the task into smaller, sequential steps, allowing for scalability and maintaining the coherence of the output. The MapReduce chain not only overcomes the limitations of the context window, but also ensures that the analysis can be applied to any codebase, regardless of its length.

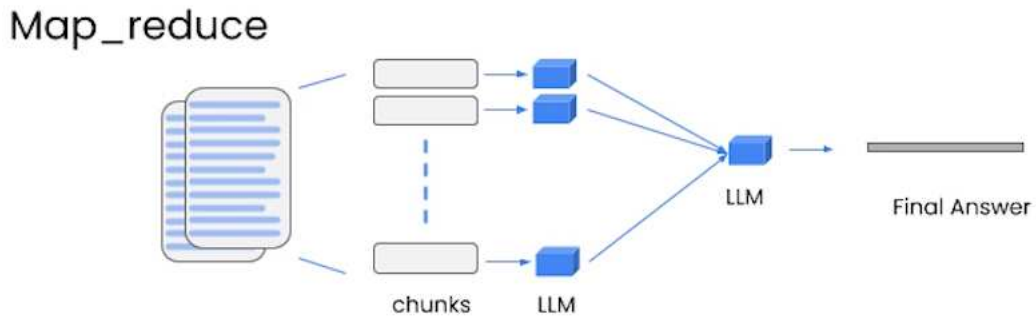

Figure 4.1: MapReduce Chain Schema [37]

We also conducted tests with another chain called Refine. Unlike MapReduce, the Refine chain processes the input in a sequential manner. It splits the documents in chunks and first passes the first one through the model to generate an initial output, then it refines this output in subsequent passes, processing the other chunks, allowing for more detailed and coherent final results. However, when choosing between the two options, MapReduce demonstrated a higher processing speed and faster output generation. For this reason, we opted for the integration of MapReduce chain in our application.

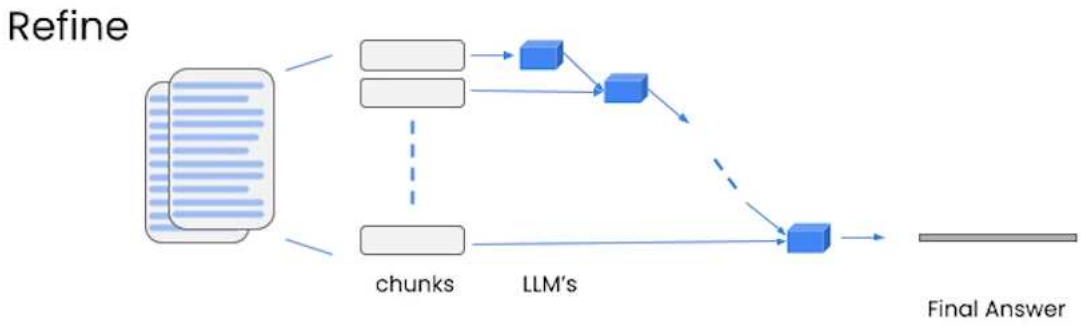

Figure 4.2: Refine Chain Schema [37]

4.4.2 Suitable Models

Despite chains being integrable wherever there is a possibility that the input exceeds the context window, their best use is when the context window is limited. MapReduce allows testing models with a smaller context, such as CodeLlama or CodeStral, while also providing an "escape condition" to prevent output production from stalling. The chain will be incorporated into the application and its performance evaluation will be conducted for the outputs where it has been used.

4.4.3 Prompting

The prompting for the MapReduce chain follow a standard process and it is strictly linked with its functioning: a summarize prompt for the Map, a combine prompt for the Reduce. In the Map prompt, the model is simply asked to summarize the contents of the assigned chunk, while in the combine prompt the summaries are combined and the assigned task executed. For illustrative purposes, the standard prompts for a textual summarization task with MapReduce are reported. Those prompts can be modified and tailored to specific tasks with simple adjustements.

Map Prompt Example

The following is a set of documents

`{docs}`

Based on this list of docs, please identify the main themes.

Reduce Prompt Example

The following is set of summaries:

`{docs}`

Take these and distill it into a final, consolidated summary of the main themes.

4.4.4 Evaluation

The outputs evaluated are produced using CodeQwen1.5-chat, with a restricted context window to simulate a excessively long input.

- **OQE**

  The documentation produced appears to be complete, covering all the required sections, and the formatting is consistent with the specified guidelines. The level of detail is satisfactory; however, there are no diagrams or charts accompanying the textual descriptions. The time required to generate the output is significantly longer compared to the direct approach.

- **Automated Metrics**

  The automated metrics are computed on the closed-source benchmark documentation. Given its length, the COBOL program "WS0C83" has been taken as evaluation example.

Metric	WS0C83
ROUGE - 1	0.36
ROUGE - 2	0.05
ROUGE - L	0.21
BERTScore	0.63

Table 4.4: Automated Metrics on "WS0C83" program - MapReduce Approach

The automated metrics reveal that the documentation produced by the open-source model with MapReduce chain integration, achieves good level of performance in capturing and describing the program's functionality and structure, compared to what the commercial model does. The ROUGE scores indicate moderate results in terms of overlapping single words used, though, as already seen, they are less promising when considering longer sequences of words. The BERTScore is good, suggesting that the overall meaning of the sentences is comparable to the reference. However, the automated metrics refer only to what is written in the general overview of the program, so the general quality of the documentation could differ and need further evaluations.

- **Evaluation with LLM**

  The LLM evaluation is run on the WSOC83 COBOL program. The output and the scores of the automated LLM evaluation are the following:

  1. **Completeness:** 2/3

     The documentation covers many aspects of the WSOC83 program, including its main functionalities, technical overview, and some details about data structures and queries. However, it lacks comprehensive information on some crucial aspects. The error handling mechanisms are mentioned but not detailed thoroughly. The specific validations performed on incoming messages are not fully explained. The exact structure of the messages being processed is not clearly defined. Some subroutines mentioned in the

original description are not covered in detail.

2. **Clarity:** 2/3 The documentation is generally understandable, but there are areas where clarity could be improved. The technical overview provides a good general understanding of the program's flow. However, some sections, particularly those describing data structures and queries, are presented as lists without much context or explanation. The relationship between different components (e.g., tables, subroutines) is not always clearly explained. Some technical terms are used without explanation, which might be confusing for readers not familiar with the specific system.

3. **Consistency:** 2/3

   The documentation maintains a reasonable level of consistency, but there are some issues. The overall structure of the documentation is consistent, with sections covering different aspects of the program. However, the level of detail varies significantly between sections. For example, the functional overview is quite detailed, while the data structures section is more of a list without much explanation. There are some inconsistencies in terminology, with some terms used interchangeably without clear definitions (e.g., CODA TD and Transaction Data Queue).

4. **Accessibility:** 2/3

   The documentation is reasonably accessible, but there's room for improvement. The information is presented in a structured manner with clear section headings. However, the lack of a table of contents or clear navigation structure makes it difficult to quickly find specific information. The use of technical jargon without explanations may make it less accessible to readers who are not familiar with the specific system or COBOL programming. The documentation would benefit from more visual aids, such as flowcharts or diagrams, to illustrate the program's structure and data flow.

In summary, while the documentation provides a good overview of the WSOC83 program, it could be improved in terms of completeness, clarity, consistency, and accessibility. Adding more detailed explanations, improving the structure, and including visual aids would significantly enhance its overall quality.

- **Human Evaluation**

  The human evaluation was carried out by myself and two expert developers in COBOL, who analyzed the output and assessed it according to the criteria and scores provided by us and reported in Section 2.2.5. The average scores are provided in the following table.

Metric	**Score**
Completeness	2/3
Clarity	2.3/3
Consistency	2/3
Accessibility	2.3/3

Table 4.5: Human Evaluation - MapReduce Approach

The evaluation is consistent with what obtained with LLM evaluation. The documentation is generally good, but it lacks of detail, examples, and visual elements.

4.4.4.1 Final Considerations

The use of the MapReduce chain allows for the analysis of documents of any size and length, even with models that have a limited context, making the application robust and generalizable. However, the quality of the generated documentation tends to be lower and more incomplete. Additionally, the MapReduce process significantly increases the production time, extending it from a few minutes to almost an hour, with an average of 360 seconds per each section of the documentation. In summary,

while MapReduce is a viable solution to prevent application crashes, an alternative approach would be preferable.

4.5 Multi-agent Approach

LLMs have evolved from being mere stand-alone assistants to almost autonomous agents capable of delegating tasks and adapting their behavior based on feedback from their environment or their own processes. A multi-agent system (MAS) is a system composed of multiple interacting intelligent agents. These systems facilitate complex interactions, improving both the capability and performance of AI-based workflows. Following the general trend of such systems, and inspired by the external provider *Company C*, we developed an application inspired by this approach that utilizes multiple specialized agents, each responsible for a different aspect of the documentation process. In particular, four different agents are involved, each with its own instruction:

1. **DocumentationGenerator:**

   Generates the initial documentation based on the cleaned code.

2. **Reviewer:**

   Reviews the generated documentation for accuracy and completeness.

3. **Improver:**

   Improves the documentation based on the reviewer's feedback.

4. **Polisher:**

   Refines and polishes the final documentation.

These agents work together in a genetic algorithm-like process to evolve and improve the documentation over multiple generations. The application also incorporates RAG (see Chapter 5.1) with a temporary vector database to store and retrieve code embeddings, using them to further enhance the system's ability to understand

and improve the documentation. The application supports both local and remote (Claude 3.5 Sonnet) LLM options, allowing for flexibility in deployment and comparison of results.

4.5.1 Prompting

Each agent in the multi-agent approach uses specialized prompts tailored to their specific tasks:

1. **DocumentationGenerator** prompt: focuses on generating comprehensive documentation, explaining the program's purpose, main components, key variables and procedures, logic flow, and notable features.

   Generator Prompt Example

   Generate comprehensive documentation for the following {LANGUAGE.upper()} code:

   `{input_data['cleaned_code']}`

   Your documentation should:
   1. Explain the overall purpose of the program based on its structure and content.
   2. Describe the main components of the code and their roles.
   3. List and explain key variables, procedures, or functions and their purposes.
   4. Outline the main logic flow of the program.
   5. Highlight any notable features, patterns, or potential issues in the code.
   Format your response in Markdown, using appropriate headers and code blocks for {LANGUAGE.upper()} snippets.

2. **Reviewer** prompt: asks for feedback on the accuracy, completeness, clarity, and readability of the documentation, as well as suggestions for improvement.

Reviewer Prompt Example

Review the following documentation for accuracy and completeness:

```
{input_data['documentation']}
```

Provide feedback on:
1. Accuracy of the technical content
2. Completeness of the documentation
3. Clarity and readability
4. Suggestions for improvement

Format your response as a list of comments and suggestions.

3. **Improver** prompt: instructs the agent to enhance the documentation based on the reviewer's feedback.

Improver Prompt Example

Improve the following documentation based on the review feedback:
Original Documentation:

```
{input_data['documentation']}
```

Review Feedback:

```
{input_data['review']}
```

Also consider these useful code snippets that could be integrated in the document:

```
{code_snippets}
```

Provide an improved version of the documentation, addressing the feedback.

4. **Polisher** prompt: focuses on refining the overall structure, clarity, formatting, and adding any missing important details.

Polisher Prompt Example

Polish and refine the following documentation:

```
{input_data['documentation']}
```

Focus on:
1. Improving overall structure and flow
2. Enhancing clarity and conciseness
3. Ensuring consistent formatting and style
4. Adding any missing important details

Provide the polished version of the documentation.

These prompts are designed to work together in an iterative process, continuously improving the quality of the documentation.

4.5.2 Evaluation

The outputs evaluated are produced using Claude 3.5 Sonnet, for closed-source results, and CodeGeeX4, for open-source results.

4.5.2.1 Closed-source results

- **OQE**

  The documentation produced through the closed-source multi-agent approach looks quite basic, but accurate. All the required sections are included and the formatting is excellent. The textual descriptions are concise, containing only the essential information needed to understand the program's functionality. This makes the documentation less verbose, cleaner and more straightforward. The time taken for the production is 1 minute and the cost is about 5$ for each document.

- **Evaluation with LLM**

  The LLM evaluation is run on the UTIP0060 COBOL program. The output and the scores of the automated LLM evaluation are the following:

1. **Completeness:** 3/3

   This documentation provides a comprehensive overview of the program. It covers the program's purpose, structure, key variables, main logic flow, limitations, usage example, testing recommendations, and potential future enhancements. The inclusion of code snippets for key variables adds to its completeness.

2. **Clarity:** 3/3

   The documentation is very clear and well-organized. It uses a logical structure with numbered sections and subsections, making it easy to follow. The language is concise yet informative, and technical terms are used appropriately. The inclusion of code snippets and a usage example enhances clarity by providing concrete illustrations.

3. **Consistency:** 3/3

   The information presented in the documentation is consistent throughout. The descriptions of variables, program flow, and functionality align with each other and with the program's stated purpose. There are no apparent contradictions or inconsistencies in the information provided.

4. **Accessibility:** 3/3

   The documentation is highly accessible. It uses clear headings and subheadings to organize information, making it easy to navigate. The language used is appropriate for technical documentation while remaining readable. The inclusion of both high-level overviews and specific details caters to different levels of technical expertise.

In summary, this documentation for UTIP0060 is of excellent quality. It provides a comprehensive, clear, consistent, and accessible description of the program. The document strikes a good balance between brevity and detail, covering all key aspects of the program from its structure and functionality to its limitations and potential improvements. The inclusion of code snippets, a

usage example, and sections on limitations and future enhancements adds significant value. These elements not only help in understanding the current state of the program but also provide insights into its potential evolution and areas for improvement. This level of documentation would be very useful for developers working with or maintaining the code, as well as for project managers or other stakeholders needing to understand the program's capabilities and limitations. The testing recommendations and future enhancements sections are particularly valuable for long-term maintenance and development planning.

- **Human Evaluation**

  The human evaluation was carried out by myself and two expert developers in COBOL, who analyzed the output and assessed it according to the criteria and scores provided by us and reported in Section 2.2.5. The average scores

Metric	**Score**
Completeness	2.7/3
Clarity	2.7/3
Consistency	2.7/3
Accessibility	2.7/3

Table 4.6: Human Evaluation - Multi-agent Approach with Closed-source model

The results, as for LLM evaluation, are great. The documentation has excellent quality, it covers all the parts of the program in a clear, accessible and concise way, providing examples and even tips for potential improvements.

4.5.2.2 Open-source results

- **OQE**

  The documentation generated using the open-source multi-agent approach is consistent with what was achieved with the closed-source models. All required sections are included, and the formatting is excellent. The content aligns closely

with previous results, with only minimal differences. Here too, the output is more concise and to the point. The time taken for the generation is not that different from the closed-source counterpart.

- **Automated Metrics**

  The automated metrics are computed on the closed-source benchmark documentation. The programs "SYSPTAB5" and "UTIP0060" have been taken as evaluation example.

Metric	SYSPTAB5	UTIP0060
ROUGE - 1	0.51	0.52
ROUGE - 2	0.26	0.23
ROUGE - L	0.34	0.41
BERTScore	0.57	0.78

Table 4.7: Automated Metrics on "SYSPTAB5" and "UTIP0060" programs, computed against closed-source benchmark - Multi-agent Approach

  The automated metrics indicate that the documentation produced by the open-source model, in its general overview of the program, is fully comparable to that produced by the commercial model, with no significant differences. The ROUGE scores are consistently strong, demonstrating a high degree of overlapping words and n-grams. The BERTScore is also high, confirming that the meaning closely aligns with the benchmark. In summary, there do not appear to be any notable differences in capturing the program's functionality and main features between the open-source and closed-source models.

- **Evaluation with LLM**

  The LLM evaluation is run on the UTIP0060 COBOL program. The output and the scores of the automated LLM evaluation are the following:

  1. **Completeness:** 3/3

Despite being more concise, this documentation covers all the crucial aspects of the program. It includes the program's purpose, main components, data structures, key variables, main logic flow, and notable features. It also mentions potential improvements, which adds value to the documentation. The conciseness does not detract from its completeness.

2. **Clarity:** 3/3

   The documentation is very clear and well-structured. It uses straightforward language and logical organization to explain the program's components and functionality. The breakdown of the program into its main divisions (Identification, Data, and Procedure) helps in understanding the program's structure. The "Key Variables" and "Main Logic Flow" sections provide a clear overview of how the program operates.

3. **Consistency:** 3/3

   The information presented in the documentation is consistent throughout. The descriptions of data structures and program flow align with each other and with the program's stated purpose. There are no apparent contradictions or inconsistencies in the information provided.

4. **Accessibility:** 3/3

   The documentation is highly accessible. It uses clear headings and subheadings to organize information, making it easy to navigate. The language used is appropriate for technical documentation while remaining readable. The concise nature of the document actually enhances its accessibility, as it provides a quick yet comprehensive overview of the program.

In summary, the documentation is of excellent quality. It manages to provide a comprehensive, clear, consistent, and accessible description of the UTIP0060 program without unnecessary verbosity. The document covers all key aspects of the program, from its structure and data components to its main logic and potential improvements. The concise nature of the document is indeed not

a negative aspect; rather, it enhances its usability by providing a quick yet thorough understanding of the program. This level of documentation would be very useful for both developers needing to work with or maintain the code, and for other stakeholders needing to understand the program's functionality.

- **Human Evaluation**

  The human evaluation was carried out by myself and two expert developers in COBOL, who analyzed the output and assessed it according to the criteria and scores provided by us and reported in Section 2.2.5. The average scores are provided in the following table.

Metric	**Score**
Completeness	2.7/3
Clarity	2.7/3
Consistency	2.3/3
Accessibility	2.7/3

Table 4.8: Human Evaluation - Multi-agent Approach with Open-source model

The results are once again excellent. As for the closed-source output, the documentation has obtained top marks for every criteria. The considerations are the same made for the closed-source documentation as the document is comprehensive, clear, accessible and concise, with examples and dedtailed explanations.

4.5.2.3 Final Considerations

The functional documentation generated through the multi-agent approach is likely the best achieved during the experimentation. It combines precision and completeness with readability and immediacy. Thanks to the agents that improve, refine, and polish the initial documentation, the result is a clean text that still includes all essential details. Moreover, the introduction of a small form of RAG, to retrieve code snippets, enhance the quality of the document, providing useful examples to

the users and demonstrating the potential of such technique. The results indicate that while the commercial LLM produces slightly more readable documentation and aligns better with the original code, the differences with the open-source model are relatively minor. This suggests that the local LLM, with only 9B parameters and a smaller context window (128K tokens vs. 200K tokens), can perform competitively against the state-of-the-art closed-source model when using this agentic approach. The multi-agent approach showcases the potential for creating a more robust and iterative documentation generation process by leveraging multiple specialized agents to produce high-quality results. Additionally, the ability to use either local or commercial LLMs offers flexibility in balancing performance against cost considerations.

4.6 Summary and Comparison

In this section, we provide a comprehensive summary of the results obtained from our experiments with various approaches, along with a detailed analysis and commentary. The results from both human and LLM evaluations, as well as the costs and time required to generate the outputs, are summarized in Table 4.9. This table allows for an immediate comparison of all the scores obtained across different approaches and model types.

Approach	**Model**	**LLM Evaluation**	**Human Eval. (Avg.)**	**Time**	**Cost**
Direct	Closed-source	**12/12**	10.4/12	**<1min**	$2-5
Direct	Open-source	9/12	9.3/12	5min	**Free**
Chain	Open-source	8/12	8.6/12	50min	**Free**
Multi-agent	Closed-source	**12/12**	**10.8/12**	**1min**	$5
Multi-agent	Open-source	**12/12**	10.4/12	10min	**Free**

Table 4.9: Summary of scores and comparison across different approaches

As evident from Table 4.9, the multi-agent approach consistently received the

highest ratings from both LLM and human evaluators, closely followed by the direct approach using a closed-source model. This suggests that the multi-agent system's ability to iteratively refine and improve the documentation leads to superior results. Interestingly, while the LLM evaluations generally align with human assessments, there is a notable tendency for the LLM to overestimate the quality of outputs, particularly for the multi-agent and direct closed-source approaches. This discrepancy highlights the importance of incorporating human evaluation in the assessment process, as human experts tend to be more discerning and cautious in their scoring. It's crucial to note that the human evaluation scores represent an average from three different evaluators. This methodology inherently makes it more challenging to achieve perfect consensus, which explains why even the highest-rated approaches don't achieve perfect scores. The variation in human scores underscores the subjective nature of documentation quality assessment and the value of multiple perspectives in the evaluation process.

Regarding time and costs, our findings reinforce the trade-offs between open-source and closed-source models:

- **Open-source models** offer the significant advantage of being free to use, making them an attractive option for organizations with budget constraints or those looking to scale their documentation efforts cost-effectively. However, this cost saving comes at the expense of longer generation times and, in some cases, a noticeable decrease in output quality.

- **Closed-source models**, while incurring a cost, demonstrate superior performance in terms of both speed and quality. The direct and multi-agent approaches using closed-source models consistently produce high-quality documentation in under a minute, making them ideal for time-sensitive projects or large-scale documentation efforts where efficiency is crucial.

- The **chain approach**, while free, shows the longest generation time and the lowest quality scores. This suggests that while it may be useful for handling

extremely long inputs, it may not be the most efficient choice for regular documentation tasks.

The following Table 4.10 provides a summary of the automated metrics scores across different approaches:

Approach	**ROUGE-1**	**ROUGE-2**	**ROUGE-L**	**BERTScore**
Direct	0.40	0.10	**0.27**	**0.62**
Chain	0.36	0.05	0.21	**0.63**
Multi-agent	**0.52**	0.25	**0.38**	**0.68**

Table 4.10: Summary of automated metrics and comparison across different approaches

The automated metrics provide valuable insights into the performance of different approaches:

- The **multi-agent approach** consistently achieves the highest scores across all metrics, particularly in ROUGE-1 and ROUGE-L. This suggests that this approach produces documentation that most closely aligns with the reference in terms of both content and structure.
- The **direct approach** performs well, especially in ROUGE-L and BERTScore, indicating good performance in capturing longer sequences and semantic meaning.
- The **chain approach**, while showing lower ROUGE scores, maintains a competitive BERTScore. This suggests that while it may not capture exact phrasing as well as other methods, it still preserves the overall semantic content effectively.

These automated metrics corroborate our qualitative assessments, demonstrating a good degree of similarity between open-source and closed-source outputs in understanding and summarizing the content of the analyzed programs. The consistently

high BERTScores across all approaches indicate that, regardless of the method used, the generated documentation maintains a high level of semantic relevance to the reference.

In conclusion, while each approach has its strengths, the multi-agent system using closed-source models emerges as the most balanced solution, offering high-quality output, fast generation times, and strong performance across all evaluation metrics. However, the choice between approaches should ultimately be guided by specific project requirements, balancing factors such as budget constraints, time sensitivity, and the desired level of output quality.

Chapter 5

Further implementations

5.1 Enhancing Legacy System Documentation: From Insights to Innovation

In Section 2.1, we discussed about the functional documentation and decision making output, as a side task we explored during our experiments. In fact, by providing a different set of prompts, the deployed applications' capability are extended beyond basic functional documentation, giving not only comprehensive and insightful analyses of legacy code programs, but also offering strategic recommendations for potential rewriting or decommissioning of applications and functionalities. The generated documents are well-structured and exhaustive, covering all critical aspects of the code, including inputs, processing, outputs, supported functional areas, and dependencies. Notably, they include well-justified and clearly explained recommendations for program modifications or rewrites, adding significant value for future planning. While the current output is impressive, as for the main application, incorporating representative code snippets could further enhance the concrete insights into program structure and style. Overall, the application's ability to generate comprehensive documentation for legacy programs is consistently high and the inclusion of decision rationale and recommendations for future development adds strategic value to the documentation. This approach may significantly accelerate the process

of knowledge transfer and system modernization.

An additional, innovative application of this technology lies in its potential to harmonize disparate legacy systems. For instance, when documenting different applications from various legacy systems that serve the same purpose (such as Euronext Securities' three distinct local applications for managing National Numbering Agency activities), the LLM can compare the functional documentation and suggest requirements for a unified application. This approach could effectively consolidate functionality without losing critical features from any of the original systems. As mentioned, initial tests of this comparative analysis have yielded impressive results, demonstrating the LLM's ability to synthesize information across multiple legacy systems and propose coherent, comprehensive specifications for a new, unified application. This capability significantly reduces the workload for systems analysts and provides a solid foundation for writing functional specifications for new applications.

Looking ahead, this technology opens up exciting possibilities for legacy system modernization. Future enhancements could include:

1. Automated code quality assessment and refactoring suggestions
2. Integration with version control systems for historical context
3. Interactive documentation that allows developers to query specific aspects of the codebase
4. Predictive analysis of potential system vulnerabilities based on documented code structures

By continually refining and expanding these capabilities, Euronext Securities can transform the challenge of legacy system documentation into an opportunity for innovation and strategic system improvement.

5.2 Retrieval-Augmented Generation

RAG, or Retrieval-Augmented Generation, enhances the capabilities of LLMs by integrating retrieval mechanisms with generative models. This methodology augments the LLM with a retrieval system that accesses an external knowledge corpus, allowing the model to produce responses that are not only accurate but also deeply informed by relevant, real-world information. Standard LLMs often generate responses that might lack depth or specific knowledge. In contrast, with RAG, when a query is made, the system first retrieves relevant information from a large dataset or knowledge base, which then informs and guides the generation of the response. This approach essentially involves two components: a retriever and a generator. The retriever identifies relevant documents or information that can help answer the query. There are different types of retrievers, such as dense retrievers, which use neural network-based techniques to create dense vector embeddings, and sparse retrievers, which rely on methods like BM25 or TF-IDF. Dense retrievers are more computationally intensive but can capture deep semantic relationships, while sparse retrievers are faster and excel at matching specific terms. The generator component, typically an LLM, produces the output using the context provided by the retriever to create a more informed and accurate response.

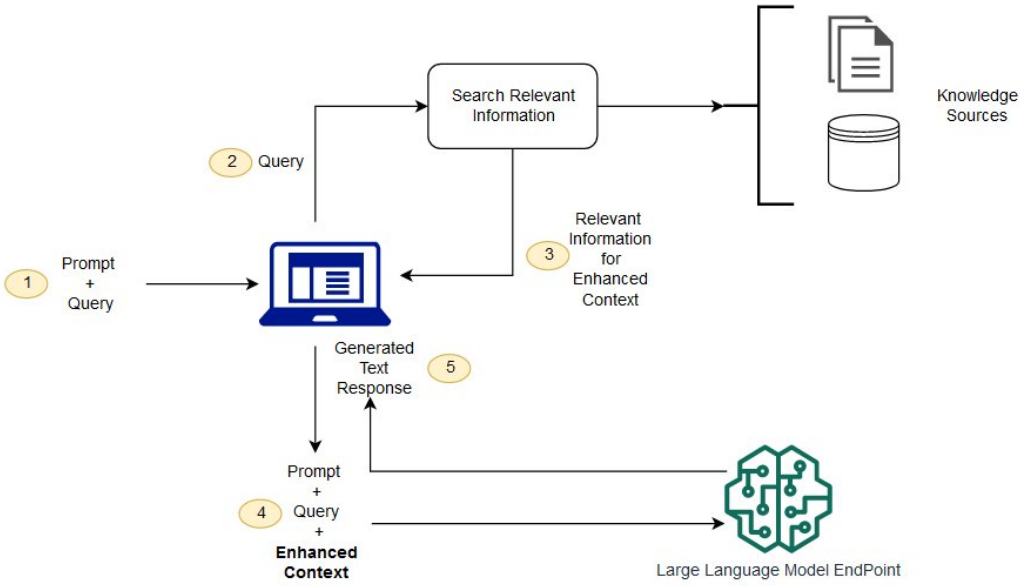

Figure 5.1: How RAG works [40]

RAG applications have quickly become the go-to solution for many business cases, primarily due to their ability to leverage new data effectively. As discussed in Section 4.5, we have already experimented with integrating the RAG paradigm into our application to enhance documentation with relevant context drawn from code snippets. The results in generating documentation are excellent, but the RAG architecture is highly expandable. The ultimate goal could be to establish a knowledge source comprised of both the code and the generated documentation, which an LLM, functioning as a chatbot, could access to answer specific user questions about programs and legacy applications. This would complete the understanding of these applications by providing the user with both static knowledge, derived from the generated documentation, and dynamic knowledge, facilitated by the RAG chatbot. This type of solution could be integrated into existing architectures, with connections to the cloud and possibly even directly to codebases. This would facilitate access and management of such an application.

Many other functionalities could be integrated in a complete RAG-based system:

1. **Cross-Code Snippet Retrieval:** when generating documentation for a spe-

cific function or module, the RAG system could retrieve relevant code snippets from other parts of the codebase that interact with or are similar to the current section.

2. **Historical Context:** by indexing previous versions of the code and associated documentation, a RAG system could provide insights into the evolution of the codebase, helping to explain why certain design decisions were made.

3. **Error Handling:** when documenting error handling procedures, the RAG system could retrieve information about known issues and their resolutions from a database of past incidents.

However, these are all possible enhancements which would require a lot of resources:

1. **A vector database** of code snippets, historical documentation, and relevant external resources should be created and maintained.

2. **Custom embedding models** to effectively capture the semantic meaning of both code and documentation.

3. **New prompts** that effectively handle the retrieved information to generate more accurate and contextually rich documentation.

In conclusion, implementing a full-scale RAG system would require significant investment in infrastructure and data preparation, but the potential benefits in terms of documentation quality and maintainability could be substantial for large, complex codebases like those found in Euronext's systems.

5.2.1 The first RAG-Chatbot in ESM

A preliminary implementation of a RAG chatbot has already been tested at Euronext Securities Milan. A simple RAG-based test application was developed to answer questions about textual documents. Users can access the tool, which runs on an EC2 instance, through a graphical interface built with Gradio (see Figure

5.2), where they can ask questions and receive responses from the LLM. The RAG knowledge is manually provided by uploading the relevant documents into a designated section. These documents are then tokenized, embedded, and stored in a vector database (in this case, ChromaDB). While this is a very "artisanal" first implementation, it already shows promising results and highlights the potential utility such a tool could offer in the daily work of an analyst.

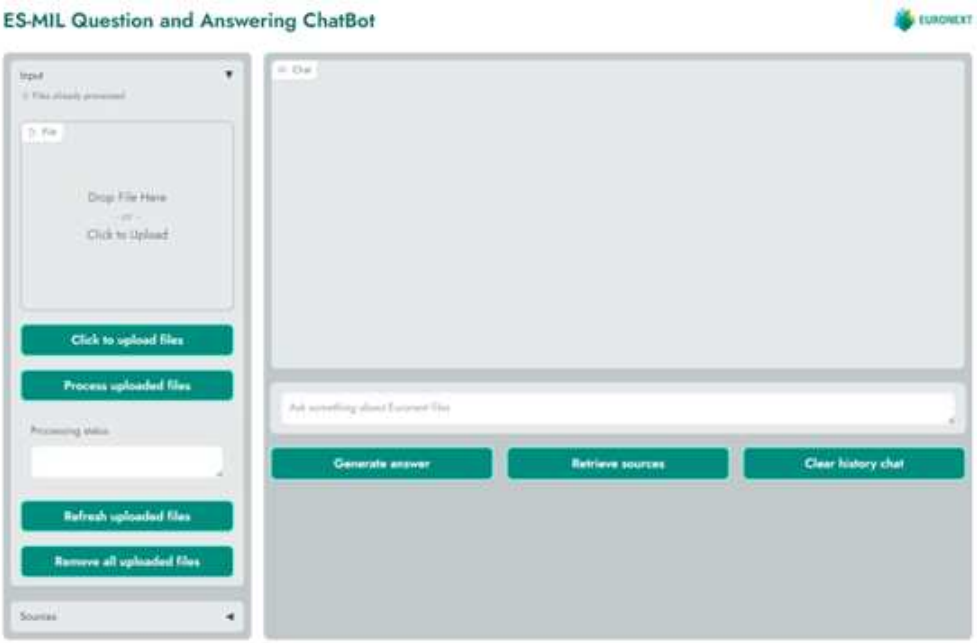

Figure 5.2: User interface of the Euronext RAG test application

5.2.2 RAG relevance

In recent months, the relevance of the RAG technique has been questioned due to the growing trend of new models featuring increasingly larger context windows. However, Retrieval-Augmented Generation remains a relevant approach, even as long-context LLMs advance in their ability to handle larger amounts of data directly within their context windows. While LLMs are increasingly capable of processing thousands or even millions of tokens, simply relying on them to retrieve and reason over extensive information can introduce challenges.

The "Needle in a Haystack" is a test created by Greg Kamradt [21] to measure the

effectiveness of LLM across various context sizes. It involves embedding specific, targeted information (the "needle") within a larger, more complex body of text (the "haystack"). The objective is to evaluate how well an LLM can retrieve and use this particular piece of information within a vast dataset. For instance, Kamradt proved how the ability of GPT-4 to retrieve and reason about multiple specific pieces of information in a large context window diminishes as the number of these pieces increases, especially when they are dispersed throughout the context. More recent and refined evaluation criteria, as the SummHay test [24], have shown similar results. This suggests that long-context models may struggle with the precise retrieval of information, particularly when multiple facts are involved. Moreover,

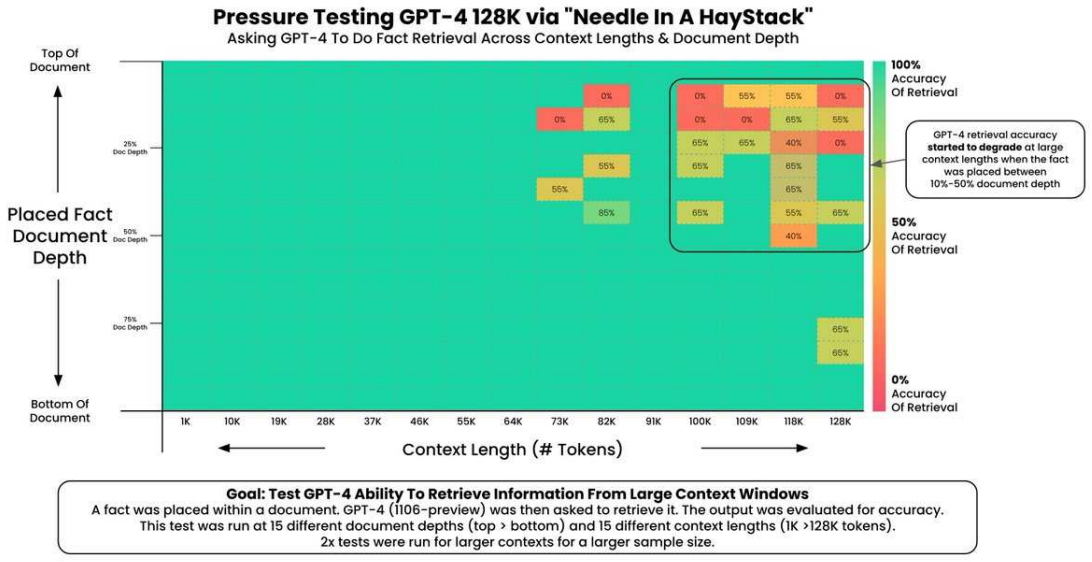

Figure 5.3: Needle in a Haystack Test on GPT-4 [15]

the phenomenon known as "recency bias", where LLMs tend to focus more on the most recent tokens, can further complicate the retrieval process when dealing with large contexts. Therefore, while long-context models are improving, the precision and reliability of RAG in extracting relevant chunks of information remain valuable. RAG excels in scenarios where specific, relevant data must be retrieved and used to generate accurate and contextually informed responses. Rather than viewing RAG as obsolete, it should be seen as evolving. A potential future direction could

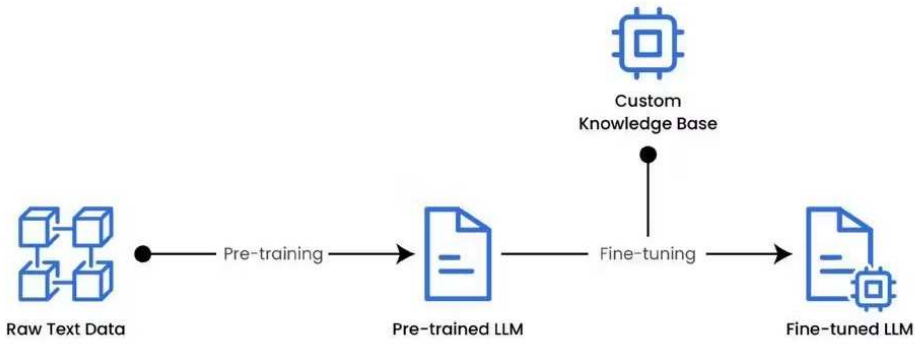

Figure 5.4: Fine-tuning schema [9]

involve a document-centric RAG approach, where entire documents are processed rather than smaller chunks, striking a balance between the efficiency of long-context LLMs and the precision of traditional RAG methods. This middle ground may offer a more effective way to manage the retrieval and generation processes as AI technology continues to advance.

5.3 Fine-tuning

Fine-tuning is a powerful process that involves taking a pre-trained model, which has already been trained on a vast amount of data, and refining it further on a specific task using a smaller, task-specific dataset. This approach is increasingly becoming a viable alternative to relying on third-party, cloud-based LLMs. Fine-tuning could represent a future implementation in the AI context within Euronext, expanding the possibilities of process automation beyond the paradigm of mainframe modernization. One of the primary advantages of fine-tuning is its potential to enhance performance and provide greater control over model behavior. By fine-tuning a base model, for instance ESM could tailor it to meet the company's specific needs, potentially surpassing the capabilities of commercial LLMs. This level of customization allows to build more robust and differentiated products, providing a competitive

edge by ensuring that the models are finely tuned to the challenges of specific tasks. Moreover, through single-task fine-tuning, it is possible to develop a collection of smaller, specialized models, each focusing on a distinct task such as documentation generation, data extraction, or code translation. This modular approach not only allows to optimize each model for its respective problem, but also helps avoid the alignment tax, where fine-tuning on one task could negatively impact performance on others. By compartmentalizing tasks, high performance across all functionalities are maintained. Fine-tuning also reduces the dependency on external services. For example, by hosting its own fine-tuned models, Euronext could minimize the legal and security risks associated with exposing proprietary data, such as personally identifiable information, internal documents, and code, to external APIs. This in-house approach mitigates concerns related to data privacy and compliance, ensuring that sensitive data remains within a controlled environment. However, fine-tuning has some big potential challenges. In fact, the process requires significant computational resources and expertise, which might pose a barrier to implementation. Moreover, fine-tuned models might not generalize as well as larger, more generalized third-party LLMs, potentially limiting their applicability across diverse tasks. Overall, fine-tuning could be a highly valuable strategy for Euronext and, carefully managing the issues of expertise and computational costs, it would be possible to create specialized, secure and efficient models that align closely with the organizational goals.

Chapter 6

Conclusion

This research demonstrates the viability and effectiveness of generative AI in producing functional documentation for legacy mainframe applications. Our experiments indicate that a multi-agent approach using state-of-the-art LLMs produces the highest quality documentation, closely followed by the direct approach. These strategies, leveraging appropriate prompting techniques and the substantial power of underlying LLMs, have shown remarkable adaptability to various input programs and consistently high output quality. The documentation generated through these approaches received excellent ratings from both human experts and LLM-based evaluations, impressing internal analysts who reviewed the documents.

While proprietary models demonstrated superior performance, our experiments also validated the feasibility of using open-source models for this task. Although the results were slightly inferior to those obtained with proprietary models, open-source alternatives showed promising potential, especially when combined with techniques like chaining to overcome context limitations. The main challenges identified with open-source models were a reduced level of detail and smaller context windows, issues that are likely to be mitigated in the near future as open-source LLMs continue to evolve. Additionally, the experiments utilized relatively small open-source models with fewer billions of parameters, and it is likely that results would improve with

larger LLMs, though these would necessitate a more robust computational environment.
Importantly, our findings extend beyond COBOL programs to include PL/1 and EGL, with experiments yielding comparable output quality across these languages. The only significant issue identified occurs in cases where code lacks inline comments, potentially necessitating human intervention for post-generation revision, However, it is scenario that should not typically happen.

The evaluation framework developed for this study proved robust and well-suited for assessing outputs. The automatic metrics, ROUGE and BERT scores, while limited in scope, provided valuable insights, and proved to be useful in comparing open and closed-source models. The LLM evaluation using Claude 3.5 Sonnet showed high correlation with human evaluation, offering detailed explanations accompanying all the scores given , as well as suggestions for improvements.

Looking ahead, several promising avenues for future research emerge:

1. **Domain-Specific Knowledge Integration:** Enhancing the documentation generation process by incorporating domain-specific knowledge bases to improve contextual accuracy and relevance.

2. **Evaluation Framework Refinement:** Developing more sophisticated evaluation metrics that better capture the nuances of code documentation quality, potentially incorporating task-specific benchmarks.

3. **Interactive Documentation Systems:** Exploring systems that can engage with developers in real-time, clarifying ambiguities and incorporating human expertise. This approach could be particularly valuable when dealing with code lacking inline comments.

4. **Long-Term Impact Assessment:** Conducting longitudinal studies to investigate the impact of AI-generated documentation on code maintainability,

developer productivity, and knowledge transfer within organizations.

5. **Scalability and Integration:** Focusing on scaling the application and integrating it into more structured, cloud-based solutions. This could involve collaboration with external providers to incorporate the application into existing enterprise environments.

6. **Advanced RAG Implementation:** Developing and integrating a sophisticated RAG-based chatbot for direct Q&A on code content, further enhancing the utility and accessibility of the generated documentation.

By pursuing these research directions and implementing the proposed enhancements, organizations like Euronext can develop innovative tools that offer significant time and resource savings, paving the way for comprehensive legacy mainframe modernization.

This experimentation has unveiled the immense potential of generative AI in code maintenance, innovation, and documentation. As more companies adopt and refine similar solutions, it's evident that in the coming years, these processes will become increasingly automated and accelerated. This progression not only ensures that critical "knowledge doesn't leave the building" but also facilitates a smoother transition from legacy systems to modern, more maintainable architectures.

In conclusion, this research represents a significant step towards automating and improving the process of legacy system documentation. By preserving and enhancing the accessibility of critical system knowledge, AI-assisted documentation tools can play a crucial role in ensuring the long-term sustainability and evolution of essential business infrastructure. As organizations increasingly face the challenge of maintaining and modernizing complex, aging systems, the approaches and findings presented in this thesis offer a valuable roadmap for leveraging AI to meet these challenges head-on.

Bibliography

[1] Meta AI. Introducing code llama, a state-of-the-art large language model for coding, 2023.

[2] Meta AI. Introducing meta llama 3: The most capable openly available llm to date. https://ai.meta.com/blog/meta-llama-3, April 18, 2024.

[3] Mistral AI. Mistral ai documentation. https://docs.mistral.ai/, April 2024.

[4] Mistral AI. Codestral: Hello, world! https://mistral.ai/news/codestral/, May 29 2024.

[5] Anthropic. Anthropic claude's documentation. https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering.

[6] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection, 2023.

[7] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru

Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report, 2023.

[8] Langchain Blog. Multi needle in a haystack. `https://blog.langchain.dev/multi-needle-in-a-haystack/`.

[9] Turing Blog. Fine-tuning llms : Overview, methods, and best practices. `https://www.turing.com/resources/finetuning-large-language-models`.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[11] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation, 2023.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[13] Democratizing Artificial Intelligence Research DAIR. Prompt engineering guide. `https://www.promptingguide.ai/techniques`.

[14] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.

[15] Aparna Dhinakaran. The needle in a haystack test: Evaluating the performance of llm rag systems.

[16] Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. A comparative analysis of large language models for code documentation generation, 2023.

[17] Wayne Xin Zhao et al. A survey of large language models, 2023.

[18] IBM. What is prompt engineering?

[19] Hongye Jin, Xiaotian Han, Jingfeng Yang, Zhimeng Jiang, Zirui Liu, Chia-Yuan Chang, Huiyuan Chen, and Xia Hu. Llm maybe longlm: Self-extend llm context window without tuning. *arXiv preprint arXiv:2401.01325*, 2024.

[20] Chandler K. Context windows: The short-term memory of large language models. navigating the basics and understanding context windows for chatgpt and llms. `https://medium.com/@crskilpatrick807/context-windows-the-short-term-memory-of-large-language-models`.

[21] Greg Kamradt. I pressure tested gpt-4's 128k context retrieval, 2023.

[22] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners, 2023.

[23] Aobo Kong, Shiwan Zhao, Hao Chen, Qicheng Li, Yong Qin, Ruiqi Sun, Xin Zhou, Enzhi Wang, and Xiaohang Dong. Better zero-shot reasoning with role-play prompting, 2024.

[24] Philippe Laban, Alexander R. Fabbri, Caiming Xiong, and Chien-Sheng Wu. Summary of a haystack: A challenge to long-context llms and rag systems, 2024.

[25] LangChain. Langchain documentation. `https://python.langchain.com`.

[26] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. Comparing code explanations created by students and large language models. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, pages 124–130, 2023.

[27] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

[28] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[29] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

[30] Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment, 2023.

[31] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work?, 2022.

[32] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding, 2024.

[33] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In Pierre Isabelle, Eugene Charniak, and Dekang Lin, editors, *Proceedings of the 40th Annual*

Meeting of the Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics.

[34] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot, 2023.

[35] Stephen Robertson, Hugo Zaragoza, et al. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389, 2009.

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

[37] Rahul S. Types of chains in langchain. `https://ogre51.medium.com/types-of-chains-in-langchain-823c8878c2e9`.

[38] Ahmed Sahin. What is retrieval-augmented generation(rag) in llm and how it works? `https://medium.com`, Apr 22 2024.

[39] Amazon Web Services. Amazon bedrock security and privacy. `https://aws.amazon.com/bedrock/security-compliance/`.

[40] Amazon Web Services. What is rag? `https://aws.amazon.com/it/what-is/retrieval-augmented-generation/`.

[41] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt, 2023.

[42] Cameron R. Wolf. The history of open-source llms. `https://cameronrwolfe.substack.com/p/the-history-of-open-source-llms-better`.

[43] Shi-Qi Yan, Jia-Chen Gu, Yun Zhu, and Zhen-Hua Ling. Corrective retrieval augmented generation, 2024.

[44] Ziyou Yan. Patterns for building llm-based systems and products. *eugeneyan.com*, Jul 2023.

[45] Shunyu Yao, Yuan Cao, and Google Research Brain Team. React: Synergizing reasoning and acting in language models. `https://research.google/blog/react-synergizing-reasoning-and-acting-in-language-models`, 2022.

[46] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.

[47] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert, 2020.

[48] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

[49] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684, 2023.

[50] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers, 2023.