innovation
for life

LLM4Legacy Study Report 2023

TNO Public ⟩ TNO 2024 R10650
19 March 2024

ICT, Strategy & Policy
www.tno.nl

TNO 2024 R10650 – 19 March 2024

LLM4Legacy Study Report 2023

Author(s)	Joe Reynolds and Rosilde Corvino
Classification report	TNO Public
Title	LLM4Legacy Study Report 2023
Report text	TNO Public
Number of pages	38 (excl. front and back cover)
Number of appendices	0
Programme number	TNO 2024 R10650
Project name	LLM4Legacy Study
Project number	060.58396/01.01

Contents

1 Introduction

This report describes the information collected between September and December 2023 during an ESI study on using Large Language Models (LLMs) to deal with legacy software. This activity aimed to build knowledge around the topic via a study of related works and first explorations with LLMs for legacy. The conclusion of this activity and, consequently, of this document is to identify a set of open research questions and a plan to address them in the coming years.

This document is structured as follows: Chapter 2 introduces the research context, explains the applied nature of the conducted research, and presents some general open questions on dealing with legacy code. Chapter 3 reports on a body of related works on LLMs, starting with a brief history of their rise and progressively diving into their use in software engineering in general and software maintenance, analysis, and restructuring in particular. Chapter 4 describes the model lifecycle and all the phases an LLM goes through to acquire general and specific capabilities. It also describes the ecosystem of LLM service providers and the frameworks and tooling used to build complex LLM-based applications as a modular pipeline. Chapter 5 describes the conducted experiments and discusses the observed results. Chapter 6 discusses general conclusions and future work.

At the end of Chapters 3, 4, and 5, we will enumerate their conclusions, using them to justify the choice of future works proposed in Chapter 6. The conclusions of each chapter refer back to parts of the text highlighted in different colors.

2 Context

ESI is a TNO department actively working with industrial partners, primarily Original Equipment Manufacturers (OEMs), for high-tech industrial equipment. We use our expertise to embed cutting-edge methodologies into the Dutch high-tech systems industry to help them cope with the ever-increasing complexity of their products.

ESI's unique role and proximity with industry is instrumental to observing and addressing cross-domain problems, which are explored and solved by mutualizing the research efforts of all our partners. Fundamental to our mission is also the tight cooperation with our academic partners. ESI's ecosystem fosters research, networking, and knowledge exchange to fulfill TNO's mission to impact industry and society positively.

A common issue troubling our industrial partners is how to deal with legacy software. Indeed, they all have systems with large, embedded code, and their machines have a long life with the support of deployed software that can last decades. The first question in this context is: what is legacy software? For some, legacy software is code, usually deployed in the field and only updated when needed. For others, it is any newly developed line of code. In both these cases, legacy software is twofold. On the one hand, it holds at its core the value and intellectual properties that give a company its competitive edge. On the other hand, it is a burden as it requires continuous evolution and maintenance.

The terms "software evolution" and "maintenance" are defined in [1] as follows:

- Software maintenance is made of preventive, corrective, or adaptive actions on deployed software (read "legacy code") to prevent it from failing, e.g., bug fixing.
- Software evolution means a continual code change during development from a lesser, basic, or worse state to an advanced or better condition.

With the advent of continuous deployment practice [2], the borders between maintenance and evolution and between deployed and development code tend to blur. Consequently, software maintenance is estimated to take up to 90% of the software development life cycle [3]. Gartner predicts that, by 2025, "technical debt will continue to compound on top of existing technical debt", consuming an even more prominent part of the current IT budget [4][5].

Despite many achievements in the field, e.g., Renaissance, Rascal, and Spoofax, dealing with software legacy is not a solved problem. At ESI, our Renaissance tools have made possible code analysis and restructuring that would otherwise have been impossible [6] [7], [8], [9]. There are, however, levels of complexity not tackled yet by our tools or other current solutions:

- The problem of (architectural) code analysis is inherently complex. We deal with much information scattered throughout the code base. Information gathering is now possible with Renaissance DB or other custom extractors. However, interpreting data, abstracting from irrelevant details, and modeling the code architecture from the collected data still requires an "expert in the loop". In this context, an open research question is the following: Is it possible to support the experts in dealing with this large amount of data and assist them in interpreting data, abstracting from irrelevant details, and modeling the code architecture from the collected data?

- Restructuring code is not always a straightforward translation from one obsolete library or design pattern to a new one. It may require complex code rewriting that also depends on the many variants of the obsolete library and design patterns occurrences in the code. In this context, open research questions are:
    - Can we help the experts identify the many variants of obsolete design patterns and libraries?
    - Can we help the expert design and implement an optimized solution for code restructuring (optimization criteria might be readability, performance, or minimum code change)?
    - Can we help the experts prove code restructuring correctness?

With these open questions in mind and following the recent successes obtained by LLMs in software engineering tasks of comparable complexity, we initiated this study to assess if and how LLMs can help us address the unresolved questions in dealing with legacy software.

3 Related Works

In this chapter, we retrace the history of NLP (Natural Language Processing) until the birth of LLMs (Large Language Models). We then focus on the successes of LLMs in software engineering in general and software maintenance in particular.

3.1 History

NLP enables computers to understand, interpret, and generate human language using a semantic model to encode and decode its meaning [10]. LLMs are part of NLP and enable many applications, such as solving speech recognition problems, transforming audio into text, automated reasoning, translation, question-answering, and text categorization.

NLP started in the 1950s and evolved throughout the 1980s, dominated mainly by the Chomskyan theories of linguistics (e.g., transformational grammar), which profoundly influenced the history of formal language parsing until recent times [11]. Transformational-grammar-based NLP is called **symbolic NLP** [12].

In the 1990s and 2000s, we witnessed noticeable successes in **statistical models**, aka methods counting word occurrences as n-grams and bag-of-words, and **prediction models**, aka methods predicting the next word in a sequence as the first artificial neural network. However, only in the 2010s, with the rise of deep neural networks, particularly **Recurrent Neural Networks** (RNN) for languages, did the prediction models start to obtain better results than symbolic NLP.

RNNs use a finite sequence to predict or label each sequence element (i.e., a word) based on the element's context (i.e., other words in the sentence). A problem with RNNs is that context information vanishes in long sequences. An evolution of RNNs that deals with this problem is **Long Short-Term RNNs**, which introduce attention mechanisms to learn what to remember and what to forget about the context of a given symbol in a sequence.

An essential complementary approach is **word embeddings**, which capture semantic features of the words and encode them as vectors in a vector space. The most straightforward form of embedding uses a dictionary as a base vector to represent words. However, with time, more and more information has been included in vector embeddings, such as position in the sentence or semantic relatedness or similarity [13].

In 2017, the paper "Attention is All You Need" [14] introduced the **transformer architecture** where only attention mechanisms are used. Such an architecture provides more parallelizable, faster-to-train encoder and decoder models. It was the birth of large language models. In 2018, OpenAI introduced the decoder-only Generative Pre-trained Transformer model [15], and in 2021, GPT-3 became a game-changer for its capability to generate human-like text [16].

In November 2022, Chat GPT was released, quickly becoming "the fastest-growing consumer software application in history" [17].

3.2 LLMs for SW Engineering

In 2021, **CodeX**, a fine-tuned GPT-3 model, started a vast trend of using LLMs for software engineering, particularly code generation. Codex is the model in GitHub's **Copilot**. By providing real-time suggestions and automating routine coding tasks, Copilot has reportedly helped developers complete tasks in less than half the time compared to developers not using it [18]. Other works advise that Copilot is an asset only for experienced developers who can filter its results for correctness and optimality [19].

Surveys [20], [21] report a large corpus of works using LLMs for many software engineering applications such as code completion, code summarization, documentation, code generation, program synthesis, software requirements engineering and design, maintenance, evolution, deployment, software analytics, software engineering processes, education, and others.

Another example of tools from these works, besides Copilot, is **AlphaCode**, which, in 2022, produced a model ranking in the top 54% in competition with 5K humans. [22] The success of AlphaCode in this competitive programming domain showcases the potential of LLMs to assist in routine coding tasks and contribute creatively to formulating solutions to novel and complex problems.

StarCoder, released in May 2023, is another LLM designed explicitly for coding applications. It is known to be a highly versatile model, given that its training data features over 80 programming languages. The model specializes in code generation and completion rather than chat, making it thrive as an AI coding assistant [23]. StarCoder's extensive context limit of 8000 tokens (small units of text) and ability to handle significant batch inferences allow it to operate at a scale greater than most other LLMs. Starcoder also aims to improve security and intellectual property protection using training data pruned only to contain permissively licensed code from GitHub.

Code LLaMa is a series of open-source LLMs produced by further fine-tuning the LLaMa 2 architecture on code and released in August 2023 [24]. It comes in three sizes: big, medium, and small, and three flavors: code, instruct, and Python. The size must be tailored based on the task and available resources. The code and code–Python models specialize in real-time code completion, similar to StarCoder. Meanwhile, the code–instruct model best understands human prompts and generates relevant code, creating a ChatGPT-like experience.

It is valuable to distinguish the open-source LLMs StarCoder, AlphaCode, and the LLaMa series from closed-source tools like ChatGPT and CodeX.

Open-source LLMs are those whose model weights are publicly available. The datasets used to train these models and architecture are also open in some cases. Consequently, a researcher could theoretically reproduce them, and they are far easier to host locally or on cloud-based services. Fully open development allows for community research, democratizes access to the models, and enables full audits throughout the whole development process.

Meanwhile, although OpenAI has made their LLMs available to the public, they have done so through a paid API service without sharing all the details regarding their development process. While API access allows researchers to experiment with these models, their ability to understand the inner workings is limited. The high development costs make it nearly impossible for academic institutions to develop these models from scratch. These factors have created anxiety among academics about whether they can meaningfully contribute to breakthroughs using closed-source models [25]. Furthermore, open-source LLMs are rapidly improving in quality and size [26].

3.3 LLMs for Software Legacy

3.3.1 Code Maintenance

Large Language Models have shown the potential to impact the field of software maintenance significantly, offering intelligent solutions for repair and vulnerability detection. Consequently, the expensive and time-consuming manual labor and expertise required for legacy software maintenance could be minimized. Hypotheses in this chapter are grounded in academic research, as evidenced by the referenced scholarly articles. In most cases, they have yet to be applied in industry.

Given their ability to assist developers with various coding tasks by synthesizing quality code [27], using LLMs for automated program repair is a natural application. Initial investigations with models like Codex [28] and StarCoder have proven their ability to complete infilling tasks. In simple terms, buggy code can be removed and filled in with newly generated code that the LLM deems suitable. Techniques have been developed recently to automate this process. Alpharepair, presented in [29], [30], replaces buggy code snippets with masked tokens and then uses the CodeBERT model [31] to replace these tokens with generated code based on the context before and after the bugs. Although they return positive results, these examples rely on using the LLM to generate programs according to the token distribution without any structural or semantic understanding of the code. This lack of understanding can lead to issues such as generating infeasible tokens without consideration of types.

Traditional deterministic Automated Program Repair (APR) tools do not suffer from these challenges, meaning novel approaches have looked at combining them with LLMs to reduce LLMs' hallucinations. Integrating a traditional completion engine has allowed for the deterministic pruning of infeasible tokens suggested by the LLM and further context for the LLM, leading to a more relevant generation for the codebase [32].

Legacy systems often lack a comprehensive suite of testing, which can lead to unexpected and hard-to-diagnose behaviors. LLMs can detect these vulnerabilities by increasing scale and picking up on nuances in the code. Programs incorporating LLMs into their workflow to automatically generate many Unit Tests relevant to a specific package or repository are already in development. With the creation of these tests on a large scale and results comparable to those written by developers, time savings can be made if these were implemented on legacy systems lacking total test coverage [33].

Alternative methods focus on using a deeper understanding of the code to generate test cases, which conventional methods may overlook. The development of differential prompting – an evolution of example and "reason and act" prompting – demonstrated that an LLM can be especially useful in writing failure-inducing tests [34], ensuring a more thorough examination of the software's robustness and reliability.

3.3.2 Code Statistics

Although they have been known to produce impressive results in exams [35], the base functionality of LLMs is to predict the next token in a sequence of words. Therefore, they are not naturally suited to performing mathematics or statistical analysis [36]. Due to their deterministic nature, traditional methods for calculating the more complex metrics that can characterize a codebase, such as cyclomatic complexity, Halstead complexity, and maintainability index, are preferable.

However, this does not mean LLMs have no application in investigating code statistics. With the development of plug-ins, Wolfram Alpha has been integrated into GPT-4's workflow. If faced with a problem requiring mathematical reasoning, the model can make a query using scientific software to calculate a suitable answer [37]. This area of LLM engineering is called agent-based and involves any LLM that can use tools like calculators, search engines, or executing code [38].

In this context, it would be feasible to have an LLM, which, when asked for cyclomatic complexity, would run a premade script to calculate this value for the given code and then format the answer as a natural language response. This integration allows for more intuitive interaction with technology, giving new opportunities to gain insight for those not experienced in statistical analysis, simplifying workflows, and saving time for experts.

3.3.3 Clone Detection and Refactoring

Legacy software systems often suffer from code duplication, leading to maintenance challenges and increased technical debt. Clone detection, the process of identifying similar or duplicate code fragments, is critical in maintaining and updating these systems. Traditionally, this has been a manual and error-prone process, with deterministic systems suffering due to the vast array of edge cases.

New approaches present compelling examples of how LLMs could be deployed for clone detection. By mapping code snippets into a high-dimensional vector space, these models can identify clusters of similar code, even if they differ in syntax [39]. The approach could be instrumental in legacy systems where different programmers might have written similar functionalities in various styles, given that code that performs similar functions or structures will have comparable vector representations.
Refactoring is crucial in legacy systems to improve maintainability and readability without changing the system's external behavior. LLMs can generate recommendations for code improvement that adhere to best practices as defined by a user prompt, thus ensuring higher code quality and reducing technical debt [40].

3.3.4 Code Analysis and Documentation

Legacy code often lacks complete or proper documentation, making it challenging and time-consuming for developers to understand the system's functionality [41]. LLMs have already proven their use in code summarization [42], and a natural extension of this ability is the automated generation of documentation or recommendations for a codebase [43]. Currently, context size is the critical limiting factor for this solution, with documentation often being generated for specific chunks of code rather than a high-level project scale.

Acting on the assumption that LLMs are adept at translating natural language to source code and vice versa, the potential use case of a documentation tester presents itself. An LLM-supported system could verify the extent to which the actual implementation of the source code matches the documentation that describes it. It could write new documentation or verify the consistency and accuracy of existing documentation with the legacy code implementation. This area has little investigation, yet it appears to be promising.

An extension of code documentation generation is using LLMs to create architectural diagrams to depict the interaction between components or other structural aspects of a codebase, which may be helpful to a system architect or an engineer to reason about current code status and possible improvements. By recognizing in code the patterns that describe the interaction between components or the structural aspect under evaluation, LLMs can generate a UML diagram describing the corresponding code structure and architecture. This type of analysis has been previously explored with parser-based solutions that help gather information scattered throughout the code base but always rely on the domain expert to interpret data, abstract from irrelevant details, and model the code architecture from the collected data. By supporting the domain expert in their tasks, this process can become more time-efficient and less complex or error-prone, ultimately providing a greater value in output.

3.4 Conclusions on Related Work

Throughout 2023, LLMs began to appear in industry-grade products, marking a significant milestone in their commercial and practical adoption. Despite this progress, it is essential to note that the predominant use of LLMs remains within academic research rather than widespread industrial deployment. We can transition these models from academic concepts to fully-fledged, industry-ready services by identifying areas where LLMs excel and enhancing their capabilities.

Reviewing Chapter 3, we come to the following conclusions:

1. LLMs excel in understanding code functionality, summarization, code generation, code improvement suggestions, and generating documentation. Evidence highlighted in chapter [3.2]
2. Open-source LLMs are preferable to closed-source ones, with advantages in transparency, accessibility, and community-driven development. Evidence highlighted in chapter [3.2]
3. LLMs generative capability is best utilized alongside human input. By generating multiple suggestions, an expert can act as a validator, selecting the best solution for a situation or iteratively improving queries to create better outputs for the situation. Evidence highlighted in chapter [3.2]
4. Experiments suggest that combining LLMs with traditional deterministic methods can improve the results of automatic code repair. Evidence highlighted in chapter [3.3.1].
5. LLMs have potential in tasks that benefit from combining their **natural randomness and general coding knowledge**. These include and are evidenced by highlights in the relevant chapter:
   a. (Unit) test generation [3.3.1]

b. Clone detection [3.3.3]
c. Detecting inconsistencies between code and documentation [3.3.4]
d. Code improvement recommendations [3.3.3]

4 LLM Lifecycle and Ecosystem

This chapter describes the LLM Lifecycle from pre-training to fine-tuning and use. It also describes the ecosystem underlying this lifecycle, computing resources, data management, service providers, and cloud infrastructure.

4.1 LLM Development and Use Lifecycle

The model development and use lifecycle for an LLM like GPT-4 involves several stages, each aiming to enhance the model's performance for specific tasks or domains. Figure 1 depicts a simplified overview of this process.

The first step is pre-training the model on Web Data. In this phase, the model is exposed to vast text data scraped from the web. The goal is to help the model learn a broad understanding of language, including grammar, vocabulary, and various writing styles. This stage does not focus on any specific task but aims to build a robust general foundation.

After pre-training, the model undergoes fine-tuning to domain or task-specific needs. This phase involves training the model on a more focused dataset relevant to a specific domain (like medical texts) or task (like answering questions or completing code). The purpose is to adapt the model's broad knowledge to the nuances and specific requirements of the targeted area.

A fine-tuned model can learn from context, which means it can understand and respond based on the immediate context provided in a prompt. This context can include instructions, examples, or specific questions. The model uses this context to generate relevant and accurate responses.

Prompts can be enhanced using templates (structured formats) or Retrieval-Augmented Generation (RAG). RAG combines the model's language generation capabilities with the ability to retrieve and use relevant external information from a database, improving the quality and relevance of responses.

One of the limitations of LLMs is the context size. The model can only process a limited amount of text at a time (like the last few paragraphs or a certain number of tokens/words).

A sliding window technique is used to manage more extended conversations or documents. Using a sliding window means the model focuses on the most recent part of the text (or conversation) and 'slides' this window forward as new information is added, effectively keeping the most relevant parts in view while older parts are phased out.

[44] provides detailed insights into the architecture and training process of GPT-3, which is similar to other LLMs.

[45] discusses the RAG system and how it enhances language model outputs with external knowledge retrieval. The following chapters discuss these concepts in more depth and summarize related works.

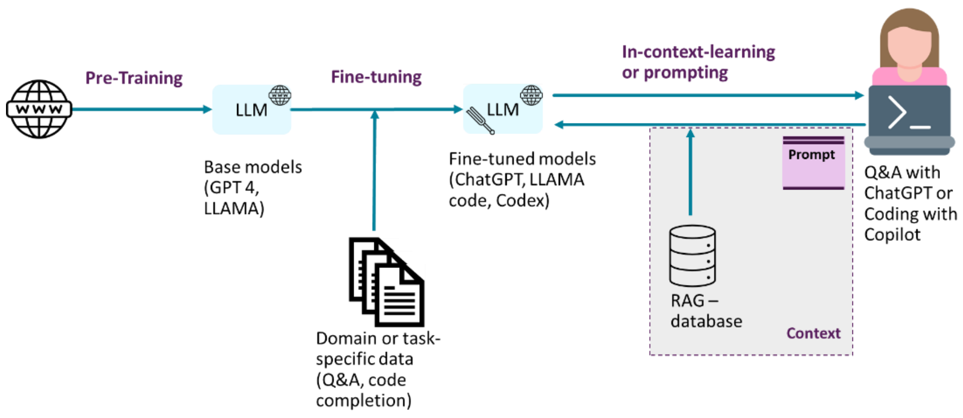
Pre-Training
LLM
Base models (GPT 4, LLAMA)
Fine-tuning
LLM
Fine-tuned models (ChatGPT, LLAMA code, Codex)
Domain or task-specific data (Q&A, code completion)
In-context-learning or prompting
Prompt
RAG – database
Context
Q&A with ChatGPT or Coding with Copilot

Figure 1. LLM development and use lifecycle

4.1.1 Pre-training

LLMs are designed to understand, generate, and manipulate language. They are 'large' as they are pre-trained on vast amounts of text data, enabling them to 'understand' a range of language patterns and idioms by predicting the next token in a self-supervised pattern [46]. This pre-training involves using vast amounts of data – usually crawled from the internet [47] - to learn billions of parameters. It is currently an incredibly expensive process in terms of both computing power and money [48]. Specific to our investigation is that codebases, including documentation and user comments, are often included in LLMs training data. Due to their ingestion of source code [23], LLMs understanding of human language and patterns can also be extended to codebases like C++. This understanding aligns well with the software engineering community's understanding of 'good code', as LLMs can grasp nuances of what engineers mean by this term due to sentiment analysis from user comments corresponding to good and bad code being included in training data. Comparisons with code quality checkers like Lint and its modern variants have shown that LLMs can contribute valuable insights. However, due to their random nature, they are not replacements for these tools and should be viewed as complementary [20].

It is currently outside the scope of this investigation to train a model from scratch due to the extreme costs. However, it will be valuable to keep up with advancements. Competitive models are rapidly decreasing in size while increasing performance, and the best models today will be easily surpassed in six months.

4.1.2 Fine-tuning

Fine-tuning is a process where a pre-trained LLM is further 'tuned' to perform a new but related task. This process readjusts the model's internal weights. The adjustments can be applied to the entire model or selectively to specific network layers. Fine-tuning can add new knowledge to a model but is primarily used to adapt its behavior [49]. The 'fine-tuned' model is an entirely distinct and specialized version of the original, so it must be hosted separately.

A common misconception regarding fine-tuning is the form of data required to complete the task. It is not enough to only have a dataset like a book or codebase to tune an LLM, at which point it will absorb the knowledge. The data must be prepared in a way that presents the expected input and output format so the model can learn the patterns. For instance, a pre-trained model is fed with question-answer pairs to train a chatbot to answer questions. Therefore, if one wanted to teach a model about a codebase using fine-tuning, one would have to give it questions regarding that codebase on the generally accepted scale of 10,000 to 1,000,000 examples. This range provides a balance between having a sufficiently large dataset to capture the nuances of a task and being manageable in terms of computational resources. This knowledge is then static and will not adjust again without retraining. Furthermore, it is 'baked into' the model, meaning recovering the source of this information becomes elusive, and it becomes difficult to check whether it is accurate and correct.

Even the most novel ways of fine-tuning LLMs remain resource-intensive [50], both in terms of computational power required for training and the storage space for multiple specialized models. When a model is fine-tuned for multiple tasks, there is the added complexity of ensuring that these tasks do not negatively influence each other – this could lead to a situation where multiple models are required for optimal performance.

Fine-tuning thrives when preparing powerful models for specific behavior-based tasks. The Llama series is a strong example of how a base model can be tuned for multiple uses [51].

- Llama 2 is a robust base model performing well in generalist benchmarking challenges.
- Code Llama is a fine-tuned version of Llama 2, better at predicting the next token in programming contexts. This model was created by fine-tuning Llama 2 on code-specific portions of its dataset for a longer time.
- Code Llama – Python was specialized by fine-tuning the Code Llama model on 100B tokens worth of example Python questions and answers. Therefore, it performs very well at tasks regarding the language.
- Code Llama – Instruct was specialized by fine-tuning Code Llama using natural language instruction input and the expected output. Consequently, it is best in conversational contexts.

Our investigation is explicitly focused on C++ understanding and, as an extension, UML or GraphML architecture diagram generation. Consequently, it would be realistic to fine-tune an LLM for one of these tasks using a dataset of C++ and UML questions and answers and then observe whether this can increase the value of our results.

4.1.3 Prompt Engineering

The prompt (input query) to an LLM can be crafted to effectively guide and optimize the generated response. Prompting involves understanding the nuances of how these models interpret and respond to different types of input and using this insight to formulate clear, contextually appropriate prompts aligned with the desired output. Effective prompt engineering can be implemented at a low cost and significantly enhance the quality, relevance, and accuracy of the responses from an LLM [52].

Prompt engineering can be split into several different sub-categories that serve various use cases and can significantly impact the effectiveness and quality of a model's response. These methods are not independent and can be combined for optimal results. Methods include:

1. Instruction-Based/Zero-Shot – Prompts are created that directly tell the model what to do. The goal is to be straightforward and unambiguous in communicating the task to the model. For example, "Write a summary of the following text..." or "Write documentation for the following code..." [53].
2. Conversational Prompting/Few-Shot – Prompts are structured as part of a conversation or dialogue and effectively guide the model to a desired response. Most chatbot applications automatically operate on this principle, where a natural conversation can guide the model to a conclusion [54].
3. Role-Based – Prompts assign the model a role, which affects the quality and style of the output. For example, "You are a helpful teacher. Please explain this concept to a child..." or "You are an experienced software engineer suggest improvements for this code..." [55].
4. Example-Based – Prompts contain examples of an ideal output that guides the model on how to respond. For example, the skeleton structure of an architectural diagram can be included to shape the output, or code can be provided to identify differences and issues [34].
5. Chain of Thought/Reasoning and Acting – Prompts give precise instructions explaining the chain of reasoning and steps the LLM should follow to achieve the desired output. The prompts also encourage the model to think aloud or detail its thought process step by step. This method is most helpful in making complex reasoning tasks more explainable to the user and improving the accuracy of the results [56].
6. Contextual – Prompts that add relevant context or background information regarding a question can help an LLM understand and generate more accurate and relevant responses [57].

Finding the optimal prompt can prove very challenging due to the sensitivity of LLMs and often requires extensive trial and error methods [58]. New techniques are being developed using predetermined systems or LLMs to give the optimal prompt based on user queries [59].

4.1.4 Retrieval Augmented Generation (RAG)

A development on contextual prompting, Retrieval Augmented Generation (RAG) integrates database searches into the LLMs' response [45]. With relevant data in the context window, the LLMs' responses are helpful, factual, and citable outputs. Various databases, including SQL, vector embedding, and knowledge graphs, can be employed.

Storing and retrieving data as vectors via apposite libraries such as FAISS or VectorDB is cost-effective and efficient compared to fine-tuning LLMs. This approach is especially beneficial for frequently updated knowledge bases, like code repositories, which receive regular push changes. The quality and completeness of the retrieved data and, therefore, the LLMs' output directly depend on the database structure and quality of embeddings.

A primary limitation of RAG is the LLMs' restricted context window. The context window dictates how much information is accessible to the LLM during inference, consequently limiting the scale and scope of the knowledge that will be used to generate a response. If not all the information required to answer a question can be included in the query context, it will be lost, and the answer will be incomplete. The context window size varies per model, with Llama-2 being about 1600 tokens and GPT-4 and Code-Llama34b being an order of magnitude greater at 16,000 tokens. Context size will likely increase as models are produced with larger architectures [60]. Innovations are underway to extend these windows with attempts to give LLMs the ability to analyze entire codebases in one shot [61]. Research demonstrates that simultaneous LLM and retriever fine-tuning can also improve RAG's ability to generate answers to domain-specific queries [62].

Evaluation of RAG systems requires insight into the retrieval system's context relevancy, the LLMs' effective use of retrieved information, and the overall quality of generated content [63].

4.2 LLM Provider Ecosystem

Figure 2 depicts the Language Model Ecosystem, which encompasses:

- Cloud Service Providers: companies and platforms that offer cloud computing resources and specialized hardware like GPUs (Graphics Processing Units) and TPUs (Tensor Processing Units) for training and deploying models. They also offer data storage and management solutions, ensuring data security and accessibility. Examples include AWS (Amazon Web Services), Google Cloud, and Microsoft Azure.
- LLM service providers like Hugging Face, OpenAI, and AWS Machine Learning. They offer platforms and tools that allow developers to access and integrate LLM functionalities into applications. More specifically, they offer APIs and tools for model selection and configuration, fine-tuning, performing inference, and efficiently mapping these models on hardware to optimize performance for various applications. LLM service providers are or make use of cloud service providers.
- Pipeline tooling: development framework offering tools for building advanced applications, using modular components as large language models, databases for retrieval augmented generation, and other agents, simplifying the creation of context-aware, reasoning AI systems.

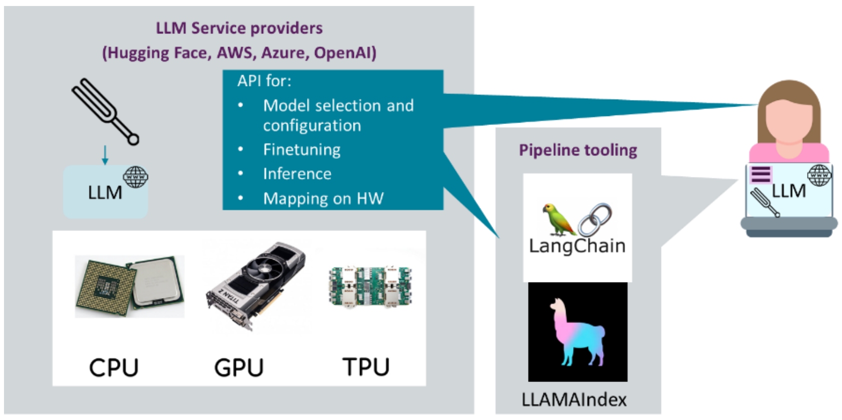

Figure 2. Language model ecosystem

4.2.1 LLM Service Providers

The massive scale of LLM models (GPT-4 has 1 trillion parameters) often makes them unwieldy and impractical to host on local devices. Consequently, several platforms have begun to offer cloud services to host custom or premade models. It is a rapidly changing landscape with a few dominant companies and new services introduced monthly.

Table 1 provides a comparative analysis of the three major service providers in the context of machine learning capabilities: Azure ML [64], AWS ML [65], and Hugging Face [66]. It aims to guide users in selecting the most appropriate service for their specific ML needs based on cost, scalability, and project requirements.

Table 1. Comparison of Cloud Service Providers

Feature/ Metric	Azure ML	AWS ML	Hugging Face
Pricing	- Pay-as-you-go - Tiered pricing based on usage - Cost to deploy Llama-2-7b inference endpoint = $7.65/hr	- Pay-as-you-go and token-based cost - Tiered pricing based on usage - Cost to deploy Llama-2-7b inference endpoint = $1.84/hr	- Pay-as-you-go - Free tier with a subscription for advanced features - Cost to deploy Llama-2-7b inference endpoint = $1.30/hr
Scalability	- Easily scalable with Azure infrastructure	- Highly scalable, integrates with AWS ecosystem	- Compute is based on an Azure and AWS provider infrastructure
Special Features	- Integrated with other Azure services	- Wide range of ML services (Bedrock is simple, Sagemaker is advanced) - Strong integration with data storage and processing services	- Focus on NLP and transformer models - Large model hub with access to the newest/most comprehensive range
Ideal Use Cases	- Enterprise solutions - Complex ML projects involving various Azure services	- Large-scale data processing and ML deployments - Projects requiring integration with AWS services	- NLP-focused projects - Researchers and small teams experimenting with pre-trained models

Azure machine learning workspaces provide a cloud-based environment where models can be trained, deployed, automated, managed, and tracked. Initially, for general machine learning models, they were quick to pivot focus to LLMs in 2023, providing premade deployment of huge LLMs. The platform has a robust infrastructure as it has been based on the standard Azure architecture, which is the basis of many businesses. Azure Kubernetes Services allows for scaling containerized ML models on a production scale. It is costly to deploy small endpoints as they only allow the renting of high-power GPUs.

Amazon Sagemaker offers a cloud environment to build, train, and deploy custom machine learning models. Very recently, in a pivot to the LLM domain, they have introduced **Amazon Bedrock,** which allows for the utilization of inference for pre-trained models, eliminating the necessity for custom training. Compute resources are priced by the hour or by tokens used/generated, giving this provider a far more flexible price point.

Hugging Face has a key focus on NLP and LLM development. Hugging Face provides the most advanced range of models as it hosts a massive range of open-source solutions. It also has a thriving open-source community that openly shares academic papers, models, and datasets as they are created. This provider has less background infrastructure, generally leasing it from AWS or Azure, but also allows for cheap and highly flexible endpoints. It was our primary choice for deploying the Code-Llama model series during this investigation.

4.2.2 Pipeline Tooling

Frameworks and libraries such as LangChain, Haystack, and LlamaIndex are versatile tooling designed to enhance large language models' capabilities by streamlining complex application development. They function as a modular pipeline, allowing developers to combine components like chat interfaces, search integrations, and language model functionalities. This abstraction layer simplifies the creation of sophisticated workflows, including information retrieval, conversation management, and data processing, refer to Figure 2. By leveraging these frameworks, developers can focus on designing and implementing the logic of their applications without getting bogged down in the intricacies of underlying model architectures. The result is a more efficient and accessible way to harness the power of language AI for innovative and practical solutions.

4.3 Conclusions on Model Lifecycle and Ecosystem

Reviewing Chapter 4, we come to the following conclusions:

1. **Training an LLM from scratch is currently too costly and challenging for our current use cases and therefore is outside the scope of this investigation**, we will instead be focusing on their applications. Evidence is highlighted in chapter [4.1.1].
2. **Fine-tuning an LLM to comprehend and generate information related to a specific codebase would be a misjudgment.** This type of fine-tuning requires input-output pairs of source code and associated questions. The major challenges are the feasibility of acquiring an appropriate dataset and then keeping this and the model updated. Evidence is highlighted in chapter [4.1.2].
3. **Fine-tuning an LLM for translating C++ source code into architectural models is a novel possibility.** This fine-tuning would require a dataset comprising source code and corresponding diagrams, which could potentially be generated using tools like Renaissance. Evidence is highlighted in chapter [4.1.2].
4. **Prompt engineering is vital in applying LLMs,** and various techniques should be employed for the best results. Evidence is highlighted in chapter [4.1.3].
5. The size of their context windows limits current LLMs. Therefore, **global code analysis is unfeasible with just-in-context learning,** given that no LLM can currently scale to a codebase featuring millions of lines of code. Evidence is highlighted in chapter [4.1.4].
6. **To source and verify information output by LLMs, the use of Retrieval Augmented Generation via a database or internet search is necessary**. LLMs often provide elusive and imprecise answers, especially in domain-specific knowledge concepts. Evidence is highlighted in chapter [4.1.4].

7. In retrieval systems, the **accuracy and completeness of an LLM's output are directly linked to the quality of the RAG embeddings or database structure**. Evidence is highlighted in chapter [4.1.4].
8. From a practical and infrastructural standpoint, it is advised to use a service provider rather than go through the prohibitive and often expensive process of setting up an LLM locally or with our API. The key benefits are flexibility and scalability. Evidence is highlighted in chapter [4.2].
9. Given that the project is still in an exploratory phase, **we recommend using Hugging Face as it is the service provider most suited to research** – with the biggest number of models and the lowest costs. In an industrial-level application, the provider can easily be switched to AWS Bedrock to serve users on a larger scale. Evidence is highlighted in chapter [4.2].

5 Experimental Setup for our Investigation

Our primary goal was to determine the feasibility and optimal methods for employing an LLM to extract and analyze knowledge from a C++ codebase. Considering input data needed to answer repository-specific questions, fine-tuning an LLM for code analysis would be expensive, codebase-dependent, and quickly become irrelevant.

We used a RAG-assisted LLM with a vector database backend to create a proof-of-concept system. Figure 3 depicts the implemented pipeline and its five stages:

1. Download
   - ➢ Objective: To acquire the target repository's code and documentation in their raw forms.
   - ➢ Versatility: The system is designed as a generalist to extract data from any C++ repository. A variety of solutions were tested. Each required optimizations, but all worked at a base level.
2. Preprocessing
   - ➢ Objective: To transform the raw codebase into an optimal state for embedding into the vector store.
   - ➢ Procedure: Automated scripts separate C++ code and text. The former is tokenized and split by declarations, while the latter keeps its form and is separated by line breaks. Metadata is added to all sections, and a directory structure is produced.
3. Processing
   - ➢ Objective: To embed the processed data into a vector store, creating a searchable, indexed database that facilitates efficient query retrieval.
   - ➢ Procedure: Embedding algorithms convert the data into vector representations stored in a FAISS database.
   - ➢ Versatility: Different embedding algorithms can be implemented quickly. During our investigation, text-embedding-ada-002 and gte-base were tested.
4. Query
   - ➢ Objective: To form the ideal RAG query tailored to our codebase.
   - ➢ Procedure: Our retriever method identifies and compiles pertinent information from the database. Components for the RAG query include the original query, relevant context, and a templated instruction for the LLM, which dictates its behavior.
5. Inference
   - ➢ Objective: To generate responses to queries using an LLM.
   - ➢ Procedure: The LLM analyses the RAG query and generates a response based on the combined more relevant information in the context.
   - ➢ Versatility: This stage is adaptable and has been tested with OpenAI's GPT-4 and Meta's Code-Llama models from Hugging Face.

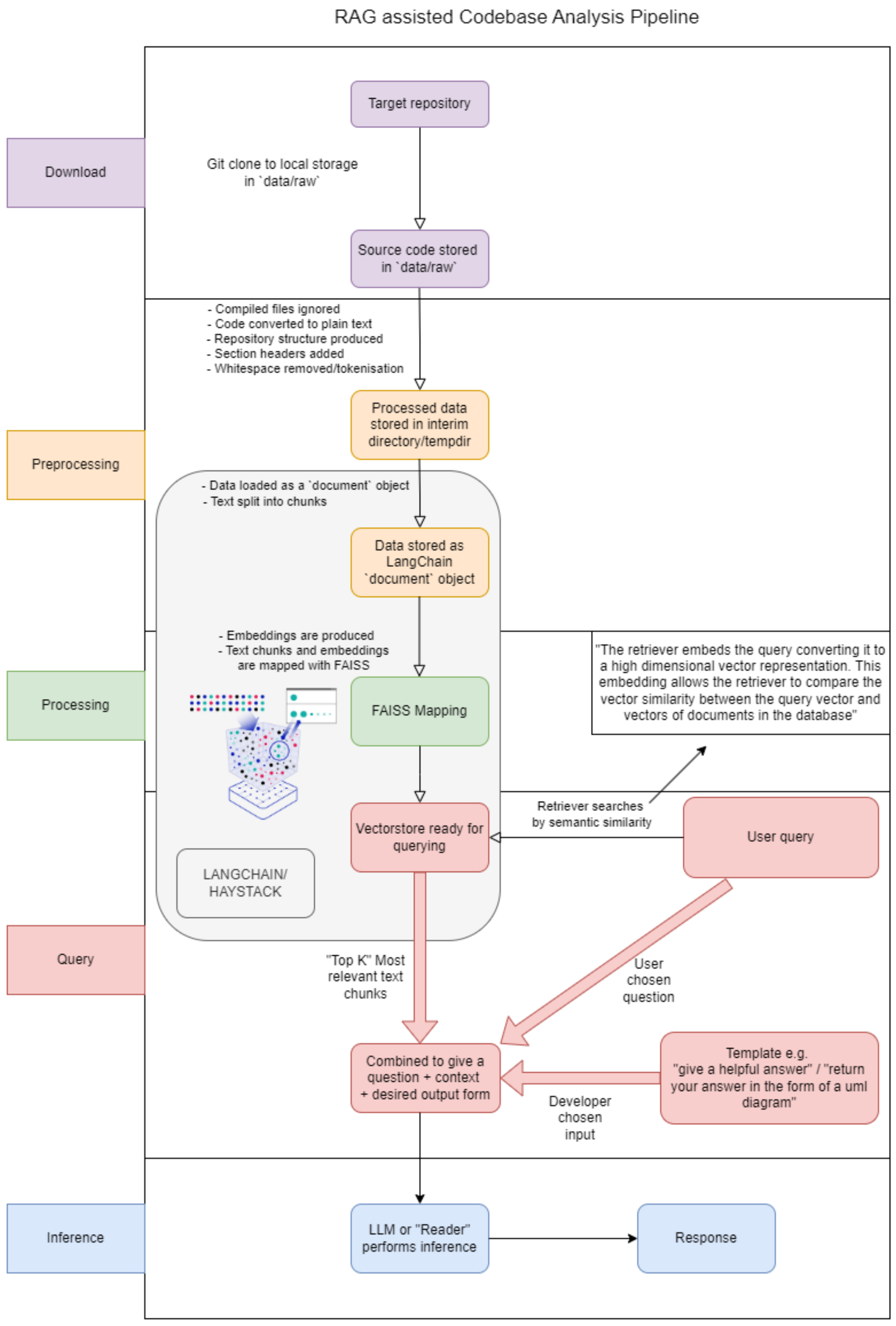
RAG assisted Codebase Analysis Pipeline
Download
Target repository
Git clone to local storage in `data/raw`
Source code stored in `data/raw`
Preprocessing
- Compiled files ignored
- Code converted to plain text
- Repository structure produced
- Section headers added
- Whitespace removed/tokenisation
Processed data stored in interim directory/tempdir
- Data loaded as a `document` object
- Text split into chunks
Data stored as LangChain `document` object
Processing
- Embeddings are produced
- Text chunks and embeddings are mapped with FAISS
FAISS Mapping
"The retriever embeds the query converting it to a high dimensional vector representation. This embedding allows the retriever to compare the vector similarity between the query vector and vectors of documents in the database"
Query
Retriever searches by semantic similarity
Vectorstore ready for querying
User query
LANGCHAIN/ HAYSTACK
"Top K" Most relevant text chunks
User chosen question
Combined to give a question + context + desired output form
Template e.g. "give a helpful answer" / "return your answer in the form of a uml diagram"
Developer chosen input
Inference
LLM or "Reader" performs inference
Response

Figure 3: Architecture of our RAG-assisted LLM pipeline

5.1 Pipeline Tooling

Our pipeline was written in Python, which features several open-source libraries for implementing LLM's into custom applications. These libraries allow for easy implementation and testing of various retrieval methods - the algorithms required to find relevant data from large databases or vector stores.

LangChain is the open-source framework we currently utilize in our pipeline. It is designed to build applications that combine LLMs with various components such as databases, APIs, and software systems. Our experience with LangChain has been positive, although it sometimes lacks modularity and is still under development, meaning it will change with time.

We also tested Haystack during our investigation. It is an open-source framework focused on building search systems for large datasets. It implements relevance scoring and a range of document retrieval methods yet suffers in its ability to interact with external APIs. This limitation made experimentation with a range of LLMs difficult, leading us to end our experiment with Haystack and continue using LangChain.

LlamaIndex is a promising new framework we look to experiment with in the future. We aim to explore whether it can complement or enhance our existing pipeline, particularly regarding data indexing, retrieval efficiency, and expansion of context windows.

Using LangChain has enabled us to effectively incorporate techniques like Prompt Engineering and RAG. As we move forward, we aim to deepen our use of these methodologies, regardless of the underlying tool, to improve the precision and context-awareness of our LLM applications.

5.2 Experiment Summary

We conducted three experiments to test the capability of base LLMs and our pipeline setup.

Table 2. Summary of experiments and results

Experiment	Remarks	Validation method	Conclusions
Architectural diagram extraction from C++ code snippets	- Base foundation models tested - Various LLMs compared - Toy code comprising Publisher Subscriber design patterns used - Limited scaling capabilities - Refined prompts required	Evaluation against Renaissance results	Renaissance outperforms out-of-the-box LLMs for diagram extraction tasks
Architectural diagram extraction from C++ repository	- Foundation models combined with RAG - Open source and obfuscated codebase analyzed - 12,000 lines of code	Evaluation against Renaissance results	Renaissance outperforms out-of-the-box LLMs for diagram extraction tasks
Generating **English explanations** from C++ repository	- Foundation models combined with RAG - Open source codebase outside the models training data - 60,000 lines of code	Evaluation against human benchmarks	Traditional tools lack these capabilities, highlighting LLMs potential in this domain

5.3 Research question 1 – Can we extract knowledge in graph form from C++ source code using LLMs and prompt engineering?

5.3.1 Introduction

This experiment delves into how LLMs, specifically GPT-4 and Code-Llama iterations, interpret C++ code, which implements a publisher-subscriber pattern. The goal is to use these generative models to create valid publisher-subscriber network graphs in dot format.

5.3.2 Methodology

The models – GPT-4, Code-Llama-7b, Code-Llama-13b, and Code-Llama-34b – were tested with five different C++ code snippets representing publisher and subscriber relationships. These snippets aimed to evaluate the LLMs ability to identify relationships in a network graph and accurately generate a diagram depicting them in a dot file.

The prompts incorporated role-based, chain-of-thought, and example-based methods to optimize results. Custom prompts were handwritten for each question. It was infrequent that a valid diagram was produced with a general prompt. As the output of these models is indeterministic each query was made three times and the best output chosen.

5.3.3 Results Summary

Table 3. **Architectural diagram** extraction from C++ code snippets

Question	GPT-4	Code-Llama-7b	Code-Llama-13b	Code-Llama-34b
1	Correct and detailed dot file. Best answer.	Failed to generate a dot diagram but demonstrated understanding through text.	Valid, but separate graphs created for each stage.	Correct dot file but uses a different approach.
2	Correct and detailed dot file. Best answer.	Failed to produce a diagram but showed understanding through text.	Correct dot file, similar to GPT-4.	Correct dot file, similar to GPT-4.
3	Correct and detailed dot file. Best answer.	Correct logic, but the diagram lacks details.	Incorrectly included arrows in both directions.	Correct and complete dot file.
4	Correct and complete dot file. Only accurate answer.	Nearly correct. Missed one connection.	Incorrect, Cluster13 was mislabelled as a publisher.	Incorrect, Cluster13 was mislabelled as a publisher.
5	Most complete and correct, with a minor artifact.	Incorrect answer.	Too general to be helpful.	Similar to GPT-4 but incomplete.

5.3.4 Conclusions

This investigation demonstrates that all our LLMs can convert source code into DOT architectural diagrams. GPT-4 performed best, but the smaller, more specialized Code-Llama-34b produced comparable results. An important note is that each question required careful prompt engineering from the researcher to produce accurate diagrams.
While this does underscore the potential for LLMs to assist in software architecture tasks such as interpretation and visualization, an automated system that consistently produces accurate results will require many more layers of complexity. A key benefit of producing these diagrams with LLMs instead of the classical approach involving parsers is that they are produced significantly faster.

Furthermore, there is a question of bias in prompt ability – prompt engineering as a science is far more developed for GPT-4 than the Llama models. Maybe with further investigation, a prompt that made the Llama models perform in a superior manner might be developed.

5.4 Research question 2 – Can we use LLMs and RAG to extract knowledge in UML diagram form from a C++ repository?

5.4.1 Introduction

This experiment assesses the capability of a RAG pipeline in generating Unified Modeling Language (UML) diagrams from a C++ codebase using natural language queries. The RAG pipeline's performance in retrieving relevant documents and accurately depicting code architecture is analyzed.

5.4.2 Methodology

The LLM was given access to information about the codebase by the pipeline described in Chapter 5.1 and a general prompt asking for a UML diagram. Natural Language Queries were made to the pipeline thrice, and the most accurate returned architecture diagram was selected.

Retriever Relevance Score indicates the efficiency of document retrieval, which varied from 1/1 (perfect retrieval) to 7/9 (suboptimal retrieval). Lower scores implied a need for refinement in the RAG search algorithm.

These diagrams were later compared with Renaissance tooling, a deterministic system that gave the ground truth. Furthermore, in some cases, the same question was asked with a text output to see whether knowledge was lost in the LLM's creation of a UML diagram.

5.4.3 Results Summary

The investigation yielded varied results across different questions, with a mix of complete and incomplete UML diagrams and varying levels of correctness and verboseness. The Retriever Relevance scores indicated the efficacy of the document retrieval process, which varied significantly across questions.

Table 2. Repository Level Architecture Extraction Results

Question	Retriever Relevance Score	Completeness	Correctness	Verboseness
Usage of `reader.h`	7/9	Incomplete	Incorrect	Acceptable
Usage of `writer.h`	17/18	Incomplete	Incorrect	Acceptable
Presence of an external library	11/17	Complete	Incorrect	Acceptable
Repository's directory structure	2/5	Complete	Correct	Acceptable
Data flow in `readFromString.cpp`	7/16	Complete	Incorrect	Excessive Information
Simplified data flow in `readFromString.cpp`	2/3	Complete	Incorrect	Acceptable
`json_cpp` header files and their usage	1	Incomplete	Correct	Excessive Information

5.4.4 Analysis of Key Questions

Q1 & Q2 (Usage of header files): Demonstrated the pipeline's partial understanding of header file dependencies but failed to represent the complete and accurate architecture in UML diagrams.
Q3 (External library algorithm): Identified most file dependencies correctly, but the UML diagram lacked accuracy and completeness.
Q4 (Repository structure): Produced a complete and correct UML diagram showcasing the RAG pipeline's potential in representing high-level structures. Due to custom embeddings for this question this is the only diagram that was perfect.
Q5 & Q6 (Dataflow in readFromString.cpp): Exhibited a mismatch between textual understanding and UML diagram representation. Both text and UML diagrams failed to describe the system accurately.
Q7 (General dependencies): While the diagram was correct, it lacked clarity due to over-complexity, underscoring the challenge of representing intricate dependencies in a comprehensible UML format.

5.4.5 Conclusions

This study reveals that the RAG pipeline can successfully retrieve relevant documents and demonstrate an understanding of C++ code. However, there are notable gaps in the pipeline's ability to accurately and consistently translate this understanding into UML diagrams. Remarkably, the pipeline struggles with general queries and representing intricate code dependencies.

One clear benefit of the LLM approach compared to parser-based methods is the speed at which it can create these diagrams. A database can be created in a matter of minutes, and each query takes less than a minute to produce a suitable graph output. Comparing this to the classical methods, which take hours to complete, a clear use case can be identified.

Limitations on current results are the RAG search algorithm, lack of structured output from our RAG search, and overly general prompts. Future improvements should focus on these aspects. Alternatively, a more specialized database – such as a knowledge graph – might be a suitable replacement for our embeddings, given that this already provides a structure, helpful metadata, and natural hierarchy.

5.5 Research question 3 – Can we extract knowledge in text form from a C++ repository using LLMs and RAG?

5.5.1 Introduction

This experiment benchmarks the ability of a RAG-assisted LLM pipeline to extract knowledge from an unfamiliar C++ codebase. By comparing human participants and LLMs, the study evaluates LLMs' understanding and response accuracy to a set of tailored software development questions at the codebase level.

5.5.2 Methodology

Humans were given access to the repository via a GitHub link. At the same time, the LLMs were connected to a vector database containing data from the repository through the RAG pipeline described in Chapter 5.1. The participants were subjected to a questionnaire designed to assess their understanding of the repository's specific functions, interactions, and broader features.

The first three questions had subjective answers. To avoid human bias, the marking process involved creating new BERT embeddings [67] for all correct non-LLM answers in phase space and finding the median point. The Euclidean distance from this point to all answers was then used to rank them, with those closest to the median being the most 'representative' or agreed upon answer, consequently being awarded the most points. The remaining questions were factual and scored based on accuracy.

It was ensured that the repository was not in the LLM's training data to emulate a setting where the experiment was conducted on proprietary software. Furthermore, LLMs were required to cite sources from the code for their responses, proving that the source of the information was retrieved.

5.5.3 Results Summary

Participants were broadly categorized into four grades based on their total scores:

- Grade A: Human-B (36), Human-A (33)
- Grade B: GPT-4 (30), Code-Llama-13b (29), Human-D (26)
- Grade C: Code-Llama-7b (23)
- Grade D: Human-C (13), Human-F (12), Human-E (10)

The results corroborated with the relative C++ skill levels of the human participants (Human-B ranked themselves as best), lending credence to our marking system. Code-Llama-13b's performance was comparable to GPT-4, demonstrating that a far smaller but fine-tuned model may be as effective in specific scenarios.

5.5.4 Analysis of Key Questions

Q1(Codebase Functionality Summary): Participants provided varying levels of detail. Human-B and GPT-4 were notably comprehensive in their answer.
Q2(Describe function 'timeSortDayList'): Humans and LLMs gave helpful answers but with interesting differences. Humans tended to give a more direct description of what the function did regarding how it interacted with the rest of the calendar. In contrast, the LLMs gave a factual documentation-like description. Code-Llama-13b was the only function to mention its dual implementation in the codebase.
Q3(Interaction between 'newEvent' and 'updateCalendar'): Humans and Code-Llamas gave more detail into the functional interplay between the two components, while GPT-4 detailed the process flow.
Q4(Where in the code is 'isDarkStyle'): Only Human-A and Code-Llama-13b identified all occurrences in the codebase. Code-Llama-13b interestingly identified its duplication at points in the repository – at one point being a 'setter' and the other a 'getter' function.
Q5(External Dependencies): The best responses identified two of three dependencies, highlighting limitations in both human and LLM abilities to locate such information.
Q6(Code smells in 'wavcat.cpp'): GPT-4 and most humans identified valid improvements, while Code-Llamas suggested only formatting enhancements.
Q7(Speech generation components): Participants were very successful, but Code-Llama excelled in detailing key components and their functions.

5.5.5 Conclusions

This study demonstrates the nuanced capabilities of RAG-assisted LLMs in comprehending and analyzing complex C++ codebases with natural language output. The larger models of Code-Llama-13b and GPT-4 performed well, particularly in detailed analysis and understanding of specific components. However, there are still some areas where human expertise outperforms, especially in identifying more general aspects of the codebase.

It is demonstrated that RAG shows promise in aiding repository-level code analysis. Nevertheless, the limitation remains that the model may not include vital facts in its answer if a critical code snippet is missed in the vector store query. However, the results still highlight the potential of LLMs in aiding software development and analysis, suggesting specific areas like documentation generation and code duplication identification for LLM-powered systems.

5.6 Conclusions on the conducted experiments

Reviewing Chapter 5, our overall conclusion is that we recommend combining multiple techniques to optimize performance. This recommendation includes the use of RAG, enhanced prompting methods, and structured embeddings to address the limitations observed and enhance the overall effectiveness of the LLMs in handling complex code-related tasks.

We split our conclusions into two categories.

Conclusions on the experimental Setup:

1) **LangChain is currently the tool of choice for creating LLM-assisted applications.** Haystack is not yet mature enough to be used, and we plan to experiment with LlamaIndex in the future. Evidence is highlighted in chapter [5.2].
2) **Code-Llama has been noted for its smaller size, cost-effectiveness, and open-source nature while delivering performance comparable to GPT-4.** This characteristic makes it viable for future exploration, although more advanced models may soon eclipse it. Evidence is highlighted in chapters [5.4] and [5.5].

Conclusions on the research questions:

3) **When provided with adequate context, LLMs demonstrate a good understanding of code and can effectively translate this understanding into natural language**. This evidence is highlighted in chapter [5.5.5].
4) **When provided with source code and specific prompting, LLMs can draw UML and GraphML diagrams**. The quality of results tends to decrease as the complexity of the code increases. This performance degradation is likely linked to the context size and quality of embeddings. Evidence is highlighted in chapter [5.3.4].
5) **RAG-assisted LLMs show enhanced code understanding and summarisation abilities at a repository level.** The effectiveness of these models heavily depends on the quality of embeddings and retrieval methods in the RAG pipeline. Evidence is highlighted in chapter [5.5.5].
6) **RAG-assisted LLMs struggle to represent code in UML diagrams.** The lack of a specific prompt, full repository-level knowledge, and structured information contribute to this problem. However, when the code directory structure was queried, the LLM performed well thanks to the independent storage of this information within the embeddings. This conclusion is highlighted in chapter [5.4.5].

6 Future Work and Report Conclusions

In exploring the application of LLMs for model-based engineering tools in C++ code, we gained significant insights and identified several avenues for future development.

The indeterministic nature of LLMs suggests that experts are still required to guide LLMs toward accurate outputs. Therefore, an LLM-based assistant that improves software architects' and developers' efficiency to achieve higher-value tasks is more realistic than a fully automated solution.
However, let us consider the following observations resulting from the research conducted so far:

- According to conclusion 1 of Chapter 3, **LLMs excel in code generation and understanding tasks.**
- According to conclusion 4 of Chapter 5, **LLMs perform well in drawing UML and GraphML diagrams with adequate context and specific prompting**. However, this performance suffers from scale.
- According to conclusion 6 of Chapter 5, **LLMs perform well if the correct information is structurally stored in embeddings**.

These observations show promise for LLMs in contributing positively to code analysis and transformation challenges.

From conclusions 3, 4, 5, and 6 of Chapter 5, we realize that **LLMs are less effective at extracting information from C++ into graphs than into natural language**. **Significant refinements in their consistency and reliability are required before they can match or surpass deterministic tools like Renaissance.** However, with these refinements, we believe that LLMs could extend or replace legacy software's current parser-based analysis and transformation methods.

Based on conclusions 3 of chapter 4 and 4 and 6 of chapter 5, a novel research area would be to **fine-tune an LLM specifically for the goal of C++ to UML or GraphML translation**. At a high level, this could be achieved by using Renaissance to create a large dataset of pairs of C++ code and corresponding architecture diagrams. This challenging experiment might fail, but it would be highly advantageous if the objective were achieved. In this case, **we could imagine producing a methodology wherein pipelines are created capable of translating from any formal language into UML or GraphML diagrams, removing, as a whole, the need to build parsers for this purpose**.

In conclusion 5, in Chapter 5, **vector embeddings are a promising method for retrieving the correct information regarding codebase-level questions;** however, while code snippets used for the **embeddings provided sufficient information for LLMs to extract textual knowledge** conclusions 4 and 6 of Chapter 5 inform us that **creating insightful, correct, and complete graphs requires more structured data and tailored prompts**.

Based on conclusion 7 of chapter 4 and conclusion 5 of chapter 5, **we propose to investigate how effective using a graph database will be as a backend for our RAG system**. We expect LLM results to be enhanced if the data can be retrieved more effectively and in a richer structure. We can also directly compare the effectiveness of semantic search and graph queries for our goals. Alternatively, based on the same conclusions, we intend to optimize our current vector store in multiple ways. **We propose to use LlamaIndex to enhance our embeddings and create multiple vector stores for specific questions**. Additionally, we **plan to make graphs using Renaissance for different architectural views, which we would annotate and use an LLM to retrieve essential data from them**.

Let us also consider the following conclusions:

- Conclusion 3, in Chapter 3, says that **LLMs' generative capabilities are best utilized alongside human input.**
- Conclusion 5, in Chapter 3, informs us that LLMs excel in:
  a) Code generation
  b) Understanding code functionality
  c) Code summarization
  d) Code documentation generation
- Conclusion 5 of Chapter 5 indicates **LLMs have potential in code improvement recommendations.**
- Conclusion 5 and 6 of Chapter 5 tell us that **RAG-assisted LLMs thrive in code understanding and summarization**; however, the quality of the embeddings in the RAG highly influences their results.

These observations suggest that LLMs can help code experts interpret data, abstract from irrelevant details, and model the code architecture from the collected data.

Building on these insights, we propose investigating how to use an LLM to assist experts in using Renaissance tools more effectively and efficiently. **We look to experiment using our LLM system to detect obsolete libraries or design patterns and offer targeted improvement suggestions.** Moreover, systems like Copilot have demonstrated the effectiveness of using an LLM to generate code within a specific template. We hope to utilize this functionality in the Renaissance system by **enabling developers to efficiently produce diverse concrete syntax patterns and fluent scripts**.

Based on our conclusions 1 of Chapter 3 and 5 of Chapter 5, which proved the effectiveness of LLMs in generating summaries of functions and components, **we aim to enrich the Neo4j database produced by Renaissance with more comprehensive information using LLM-generated documentation**.

Lastly, conclusion 5a of Chapter 3 indicates that LLMs can effectively create tests. **We propose to harness this to improve Renaissance by adding a layer of auto-generated tests to validate code restructuring before implementing significant system changes**.

In conclusion, while LLMs hold great promise as tools to augment the field of legacy software analysis and transformation, there is a clear need for further development to harness their capabilities. Their inherent indeterminism suggests that the most innovative and efficient applications will be when they are utilized alongside deterministic tools or under the guidance of experienced professionals.

6.1 Our Future Research Questions

The most achievable goals are at the top and the hardest at the bottom.

- How do we integrate LLMs, Renaissance, RAG, and prompting in an agent-based architecture?
- How can we use Renaissance to create structured embeddings for the vector store and LLM with pictures/UML and annotations, Graph search, or Neo4j integration?
- How can we use an LLM to improve the Renaissance user experience, save developers time, and further reduce the complexity of code analysis and refactoring?
    - Can we analyze code to find obsolete libraries or design patterns and suggest how to improve them? – improve human decision
    - Can we generate all possible variations of concrete syntax patterns and fluent scripts to automate the creation of renaissance code analyses and transformations? – Streamline Renaissance input
    - Can we augment a Neo4j database with summary/documentation for functions, classes, declarations, and libraries? – Improve human decisions with a better data store. Would it be preferable to write all beforehand or popup and make a small search
    - Can we use an LLM to generate tests before implementing significant system changes – improve human efficiency
- Is it possible to fine-tune a model to translate C++ into UML or GraphML diagrams? Is fine-tuning a solution to emulate Renaissance behavior with an LLM

7 Bibliography

[1] P. Tripathy, "Software Evolution and Maintenance".

[2] A. A. U. Rahman, E. Helms, L. Williams, and C. Parnin, "Synthesizing Continuous Deployment Practices Used in Software Development," in *2015 Agile Conference*, Aug. 2015, pp. 1–10. doi: 10.1109/Agile.2015.12.

[3] "Software maintenance costs," archive.ph. Accessed: Nov. 02, 2023. [Online]. Available: https://archive.ph/oBlIr

[4] "Application Modernization Should Be Business-Centric, Continuous and Multiplatform." Accessed: Nov. 02, 2023. [Online]. Available: https://www.gartner.com/en/documents/3848474

[5] "What are the drivers for application modernisation? | Computer Weekly," ComputerWeekly.com. Accessed: Nov. 02, 2023. [Online]. Available: https://www.computerweekly.com/feature/What-are-the-drivers-for-application-modernisation

[6] S. Klusener, A. Mooij, J. Ketema, and H. Van Wezep, "Reducing Code Duplication by Identifying Fresh Domain Abstractions," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Madrid: IEEE, Sep. 2018, pp. 569–578. doi: 10.1109/ICSME.2018.00020.

[7] A. J. Mooij, J. Ketema, S. Klusener, and M. Schuts, "Reducing Code Complexity through Code Refactoring and Model-Based Rejuvenation," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, London, ON, Canada: IEEE, Feb. 2020, pp. 617–621. doi: 10.1109/SANER48275.2020.9054823.

[8] P. Van de Laar, "TNO/Renaissance-Ada." TNO, Dec. 08, 2023. Accessed: Jan. 18, 2024. [Online]. Available: https://github.com/TNO/Renaissance-Ada

[9] M. T. W. Schuts, R. T. A. Aarssen, P. M. Tielemans, and J. J. Vinju, "Large-scale semi-automated migration of legacy C/C++ test code," *Softw. Pract. Exp.*, vol. 52, no. 7, pp. 1543–1580, Jul. 2022, doi: 10.1002/spe.3082.

[10] D. Khurana, A. Koli, K. Khatter, and S. Singh, "Natural language processing: state of the art, current trends and challenges," *Multimed. Tools Appl.*, vol. 82, no. 3, pp. 3713–3744, Jan. 2023, doi: 10.1007/s11042-022-13428-4.

[11] "Parsing: a timeline -- V3.1." Accessed: Nov. 03, 2023. [Online]. Available: https://jeffreykegler.github.io/personal/timeline_v3

[12] "Natural language processing," *Wikipedia*. Oct. 24, 2023. Accessed: Nov. 03, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Natural_language_processing&oldid=1181607362

[13] "A Brief History of Word Embeddings," Gavagai. Accessed: Nov. 03, 2023. [Online]. Available: https://www.gavagai.io/text-analytics/a-brief-history-of-word-embeddings/

[14] A. Vaswani *et al.*, "Attention Is All You Need." arXiv, Aug. 01, 2023. doi: 10.48550/arXiv.1706.03762.

[15] "Improving language understanding with unsupervised learning." Accessed: Nov. 03, 2023. [Online]. Available: https://openai.com/research/language-unsupervised
[16] J. Yang *et al.*, "Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond." arXiv, Apr. 27, 2023. doi: 10.48550/arXiv.2304.13712.
[17] "ChatGPT," *Wikipedia*. Nov. 02, 2023. Accessed: Nov. 03, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=ChatGPT&oldid=1183178806
[18] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The Impact of AI on Developer Productivity: Evidence from GitHub Copilot." arXiv, Feb. 13, 2023. doi: 10.48550/arXiv.2302.06590.
[19] A. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. (Jack) Jiang, "GitHub Copilot AI pair programmer: Asset or Liability?," *J. Syst. Softw.*, vol. 203, p. 111734, Sep. 2023, doi: 10.1016/j.jss.2023.111734.
[20] A. Fan *et al.*, "Large Language Models for Software Engineering: Survey and Open Problems." arXiv, Nov. 11, 2023. Accessed: Dec. 21, 2023. [Online]. Available: http://arxiv.org/abs/2310.03533
[21] X. Hou *et al.*, "Large Language Models for Software Engineering: A Systematic Literature Review." arXiv, Sep. 12, 2023. Accessed: Dec. 21, 2023. [Online]. Available: http://arxiv.org/abs/2308.10620
[22] Y. Li *et al.*, "Competition-Level Code Generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, Dec. 2022, doi: 10.1126/science.abq1158.
[23] R. Li *et al.*, "StarCoder: may the source be with you!" arXiv, May 09, 2023. Accessed: Dec. 05, 2023. [Online]. Available: http://arxiv.org/abs/2305.06161
[24] B. Rozière *et al.*, "Code Llama: Open Foundation Models for Code".
[25] J. Togelius and G. N. Yannakakis, "Choose Your Weapon: Survival Strategies for Depressed AI Academics." arXiv, Mar. 31, 2023. Accessed: Dec. 05, 2023. [Online]. Available: http://arxiv.org/abs/2304.06035
[26] H. Chen *et al.*, "ChatGPT's One-year Anniversary: Are Open-Source Large Language Models Catching up?" arXiv, Dec. 05, 2023. Accessed: Dec. 08, 2023. [Online]. Available: http://arxiv.org/abs/2311.16989
[27] J. Austin *et al.*, "Program Synthesis with Large Language Models." arXiv, Aug. 15, 2021. Accessed: Dec. 04, 2023. [Online]. Available: http://arxiv.org/abs/2108.07732
[28] J. A. Prenner, H. Babii, and R. Robbes, "Can OpenAI's codex fix bugs?: an evaluation on QuixBugs," in *Proceedings of the Third International Workshop on Automated Program Repair*, Pittsburgh Pennsylvania: ACM, May 2022, pp. 69–75. doi: 10.1145/3524459.3527351.
[29] C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-trained Language Models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, Melbourne, Australia: IEEE, May 2023, pp. 1482–1494. doi: 10.1109/ICSE48619.2023.00129.
[30] C. S. Xia and L. Zhang, "Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-shot Learning." arXiv, Jul. 25, 2022. Accessed: Dec. 04, 2023. [Online]. Available: http://arxiv.org/abs/2207.08281
[31] Z. Feng *et al.*, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages." arXiv, Sep. 18, 2020. Accessed: Dec. 19, 2023. [Online]. Available: http://arxiv.org/abs/2002.08155

[32] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Nov. 2023, pp. 172–184. doi: 10.1145/3611643.3616271.
[33] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation." arXiv, Sep. 06, 2023. Accessed: Dec. 04, 2023. [Online]. Available: http://arxiv.org/abs/2302.06527
[34] T.-O. Li *et al.*, "Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting." arXiv, Sep. 09, 2023. Accessed: Dec. 05, 2023. [Online]. Available: http://arxiv.org/abs/2304.11686
[35] C. G. West, "AI and the FCI: Can ChatGPT Project an Understanding of Introductory Physics?" arXiv, Mar. 26, 2023. Accessed: Dec. 05, 2023. [Online]. Available: http://arxiv.org/abs/2303.01067
[36] X.-Q. Dao and N.-B. Le, "Investigating the Effectiveness of ChatGPT in Mathematical Reasoning and Problem Solving: Evidence from the Vietnamese National High School Graduation Examination." arXiv, Oct. 31, 2023. Accessed: Dec. 05, 2023. [Online]. Available: http://arxiv.org/abs/2306.06331
[37] E. Davis and S. Aaronson, "Testing GPT-4 with Wolfram Alpha and Code Interpreter plug-ins on math and science problems." arXiv, Aug. 14, 2023. Accessed: Dec. 05, 2023. [Online]. Available: http://arxiv.org/abs/2308.05713
[38] Q. Wu *et al.*, "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation." arXiv, Oct. 03, 2023. Accessed: Dec. 05, 2023. [Online]. Available: http://arxiv.org/abs/2308.08155
[39] G. Malik, M. Cevik, and A. Başar, "Data Augmentation for Conflict and Duplicate Detection in Software Engineering Sentence Pairs." arXiv, May 16, 2023. Accessed: Dec. 05, 2023. [Online]. Available: http://arxiv.org/abs/2305.09608
[40] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design." arXiv, Mar. 11, 2023. Accessed: Sep. 22, 2023. [Online]. Available: http://arxiv.org/abs/2303.07839
[41] T. Ahmed and P. Devanbu, "Few-shot training LLMs for project-specific code-summarization." arXiv, Sep. 08, 2022. Accessed: Dec. 05, 2023. [Online]. Available: http://arxiv.org/abs/2207.04237
[42] M. F. Wong, S. Guo, C. N. Hang, S. W. Ho, and C. W. Tan, "Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review," *Entropy*, vol. 25, no. 6, p. 888, Jun. 2023, doi: 10.3390/e25060888.
[43] P. Bhattacharya *et al.*, "Exploring Large Language Models for Code Explanation." arXiv, Oct. 25, 2023. Accessed: Dec. 05, 2023. [Online]. Available: http://arxiv.org/abs/2310.16673
[44] T. B. Brown *et al.*, "Language Models are Few-Shot Learners," arXiv.org. Accessed: Dec. 21, 2023. [Online]. Available: https://arxiv.org/abs/2005.14165v4
[45] P. Lewis *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." arXiv, Apr. 12, 2021. Accessed: Dec. 19, 2023. [Online]. Available: http://arxiv.org/abs/2005.11401

[46] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation." arXiv, Aug. 05, 2023. Accessed: Dec. 06, 2023. [Online]. Available: http://arxiv.org/abs/2308.02828
[47] A. Al-Kaswan and M. Izadi, "The (ab)use of Open Source Code to Train Large Language Models." arXiv, Feb. 28, 2023. Accessed: Sep. 22, 2023. [Online]. Available: http://arxiv.org/abs/2302.13681
[48] X. Li *et al.*, "FLM-101B: An Open LLM and How to Train It with $100K Budget." arXiv, Sep. 17, 2023. Accessed: Dec. 06, 2023. [Online]. Available: http://arxiv.org/abs/2309.03852
[49] C. Huyen, "Reinforcement_Learning_from_Human_Feedback." Accessed: Dec. 08, 2023. [Online]. Available: https://huyenchip.com/2023/05/02/rlhf.html
[50] K. Lv, Y. Yang, T. Liu, Q. Gao, Q. Guo, and X. Qiu, "Full Parameter Fine-tuning for Large Language Models with Limited Resources." arXiv, Jun. 16, 2023. Accessed: Dec. 06, 2023. [Online]. Available: http://arxiv.org/abs/2306.09782
[51] H. Touvron, L. Martin, and K. Stone, "Llama 2: Open Foundation and Fine-Tuned Chat Models".
[52] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, "Prompt Engineering or Fine Tuning: An Empirical Assessment of Large Language Models in Automated Software Engineering Tasks." arXiv, Oct. 10, 2023. Accessed: Dec. 06, 2023. [Online]. Available: http://arxiv.org/abs/2310.10508
[53] Z. Yuan, J. Liu, Q. Zi, M. Liu, X. Peng, and Y. Lou, "Evaluating Instruction-Tuned Large Language Models on Code Comprehension and Generation." arXiv, Aug. 02, 2023. Accessed: Dec. 06, 2023. [Online]. Available: http://arxiv.org/abs/2308.01240
[54] T. B. Brown *et al.*, "Language Models are Few-Shot Learners." arXiv, Jul. 22, 2020. Accessed: Dec. 06, 2023. [Online]. Available: http://arxiv.org/abs/2005.14165
[55] M. Shanahan, K. McDonell, and L. Reynolds, "Role-Play with Large Language Models." arXiv, May 25, 2023. Accessed: Dec. 06, 2023. [Online]. Available: http://arxiv.org/abs/2305.16367
[56] J. Wei *et al.*, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." arXiv, Jan. 10, 2023. Accessed: Sep. 22, 2023. [Online]. Available: http://arxiv.org/abs/2201.11903
[57] S. Gao, X.-C. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu, "What Makes Good In-context Demonstrations for Code Intelligence Tasks with LLMs?" arXiv, Aug. 08, 2023. Accessed: Dec. 06, 2023. [Online]. Available: http://arxiv.org/abs/2304.07575
[58] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp, "Fantastically Ordered Prompts and Where to Find Them: Overcoming Few-Shot Prompt Order Sensitivity." arXiv, Mar. 03, 2022. Accessed: Dec. 06, 2023. [Online]. Available: http://arxiv.org/abs/2104.08786
[59] Q. Ye, M. Axmed, R. Pryzant, and F. Khani, "Prompt Engineering a Prompt Engineer." arXiv, Nov. 09, 2023. Accessed: Dec. 06, 2023. [Online]. Available: http://arxiv.org/abs/2311.05661
[60] "Anthropic_Introducing_Claude_2.1.pdf."
[61] F. Zhang *et al.*, "RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation." arXiv, Oct. 20, 2023. Accessed: Dec. 19, 2023. [Online]. Available: http://arxiv.org/abs/2303.12570

[62] S. Siriwardhana, R. Weerasekera, E. Wen, T. Kaluarachchi, R. Rana, and S. Nanayakkara, "Improving the Domain Adaptation of Retrieval Augmented Generation (RAG) Models for Open Domain Question Answering," *Trans. Assoc. Comput. Linguist.*, vol. 11, pp. 1–17, Jan. 2023, doi: 10.1162/tacl_a_00530.
[63] S. Es, J. James, L. Espinosa-Anke, and S. Schockaert, "RAGAS: Automated Evaluation of Retrieval Augmented Generation." arXiv, Sep. 26, 2023. Accessed: Dec. 19, 2023. [Online]. Available: http://arxiv.org/abs/2309.15217
[64] "Azure Machine Learning - ML as a Service | Microsoft Azure." Accessed: Jan. 05, 2024. [Online]. Available: https://azure.microsoft.com/en-us/products/machine-learning
[65] "Machine Learning and Artificial Intelligence - Amazon Web Services," Amazon Web Services, Inc. Accessed: Jan. 05, 2024. [Online]. Available: https://aws.amazon.com/machine-learning/
[66] "Hugging Face – The AI community building the future." Accessed: Jan. 05, 2024. [Online]. Available: https://huggingface.co/
[67] D. Dhami, "Understanding BERT — Word Embeddings," Medium. Accessed: Jan. 05, 2024. [Online]. Available: https://medium.com/@dhartidhami/understanding-bert-word-embeddings-7dc4d2ea54ca

ICT, Strategy & Policy

www.tno.nl

innovation
for life