Technische Universität Berlin

Bachelor Thesis

Applying DLR's Software Engineering Guidelines through an LLM for CI Pipeline and Software Testing

Nicolas Alejandro Dahbar Miguez

11.11.2024

Review by
Prof. Dr. Michael Felderer
Prof. Dr. Manfred Hauswirth

Supervised by
Carina Haupt, Dr. Tobias Hecking

Abstract

This thesis explores the ability of Large Language Models, specifically GPT-4, to automate key tasks in software engineering: generating continuous integration pipelines and software tests. Using the open-source GitLab-Calendar project from the German Aerospace Center (DLR) as a case study, the generated outputs are evaluated for both functionality and adherence to the DLR Software Engineering Guideline. Detailed prompts were prepared to provide the necessary context, project structure, and guideline requirements to the LLM for generating accurate responses.

The results show that while GPT-4 can quickly generate CI pipelines and software tests with proper structure, the outputs were not fully functional and did not meet all guideline recommendations. The generated CI pipeline encountered errors due to wrong assumptions about project dependencies, and the software tests were based on a misunderstanding of the project's code structure. However, GPT-4 demonstrated an understanding of industry best practices, and under human oversight, it has the potential to speed up the development process by automating boilerplate code and configuration files.

This study highlights both the strengths and limitations of LLMs in software engineering, showing the importance of iterative prompt refinement and human intervention to correct errors and optimize results. Although LLMs can provide templates and improve productivity, they are not yet capable of fully replacing human developers in complex software projects. The thesis concludes by discussing the results of the study and suggesting areas for further research.

Zusammenfassung

Diese Arbeit untersucht die Fähigkeiten von Large Language Models, insbesondere GPT-4, zur Automatisierung von zwei Aufgaben im Software Engineering: der Generierung von Continuous Integration Pipelines und Softwaretests. Als Fallstudie dient das Open Source Projekt GitLab-Calendar des Deutschen Zentrums für Luft- und Raumfahrt (DLR), und die generierten Ergebnisse werden von ihrer Funktionalität und ihrer Erfüllung der DLR Software Engineering Richtlinie bewertet. Ein detaillierter Eingabeprompt wurde erstellt, um dem LLM den notwendigen Kontext, die Projektstruktur und die Richtlinienanforderungen bereitzustellen, damit es korrekte Antworten generieren kann.

Die Ergebnisse zeigen, dass GPT-4 zwar in der Lage ist, CI-Pipelines und Softwaretests schnell zu generieren, die Ausgaben jedoch nicht vollständig funktionsfähig waren und nicht vollständig den Richtlinien entsprachen. Die generierte CI-Pipeline scheiterte aufgrund falscher Annahmen über die Projektabhängigkeiten, und die Softwaretests basierten auf einem Missverständnis der Projektstruktur. Dennoch zeigte GPT-4 ein Verständnis für bewährte Branchenpraktiken, und unter menschlicher Aufsicht kann es den Entwicklungsprozess durch die Automatisierung von Boilerplate-Code beschleunigen.

Die Studie hebt die Stärken und Schwächen von LLMs in der Softwareentwicklung hervor und betont die Notwendigkeit iterativer Verbesserungen der Eingabeprompt sowie menschlicher Aufsicht, um Fehler zu korrigieren und die Ausgaben zu optimieren. Obwohl LLMs Aufgaben automatisieren und die Produktivität steigern können, sind sie noch nicht in der Lage, menschliche Entwickler in komplexen Softwareprojekten vollständig zu ersetzen. Die Arbeit schließt mit einer Diskussion über die zukünftigen Einsatzmöglichkeiten von LLMs in der Softwareentwicklung und schlägt Richtungen für weitere Forschung vor.

Contents

1. Introduction

1.1. Motivation

This thesis aims to investigate whether LLMs can effectively generate CI pipelines and software tests that adhere to the German Aerospace Center (DLR) Software Engineering Guideline [44], analyzing the key question of whether LLMs can reliably produce CI pipelines and software tests that align with the DLR Software Engineering Guideline in an automated software development process. The DLR Guideline serves as a set of recommendations and best practices to ensure quality and consistency within software engineering.

In recent years, Generative Artificial Intelligence (AI) has emerged as a major topic of discussion in computer science, driving significant advancements in the field. Large Language Models (LLMs), in particular, are increasingly used across various software engineering applications [35], making them a very important tool to study in this field.

LLMs are advanced artificial intelligence systems capable of understanding, generating human-like text, and even producing code based on extensive training data. Due to their ability to interpret context, generate coherent responses, and even automate certain coding tasks [35], these models have the potential to reduce the manual effort required to develop and maintain many software engineering practices [3]. For instance LLMs can be used to generate boilerplate code, and even suggest code fixes based on identified bugs. This can reduce the time spent on tedious tasks, allowing developers to focus on problem solving.

1.2. Objectives of the Study

While Large Language Models offer very strong capabilities in software development [47], their use is not without challenges. LLMs, like any AI technology, can produce errors or generate responses that don't align with specific user requirements or may simply fail to work correctly in certain contexts [20]. A key limitation is their "black box" nature, meaning it is almost impossible to understand the reasoning behind their generated responses [5].

These issues give importance to a detailed study to see their effectiveness and reliability in software engineering. This thesis will evaluate the correctness of Large

Language Models outputs in CI pipelines and software testing scenarios for a given software engineering project, while also making sure it complies with the software engineering guideline from the DLR. Here we focus on assisting researchers who may not have extensive knowledge in software engineering, providing them with a tool that could improve their development process by generating configurations and tests based on established guidelines.

1.3. Scope

On this thesis we will focus on exploring the application of Large Language Models in developing Continuous Integration pipelines and software tests for a given project. This work will investigate the following aspects:

- The use of related sections from the DLR software engineering guideline as input prompts for an LLM, with the goal of generating CI pipeline configurations and software.
- The evaluation of the generated CI pipelines and software tests in terms of their correctness, completeness, and adherence to the DLR guideline.
- An analysis of the potential benefits and challenges of integrating LLMs into the software development process, including their impact on productivity and code quality.

The DLR guideline establishes quality standards and best practices for DLR scientists, aiming to improve the software they develop through good software development and documentation practices. The sections of the guideline that we will be working with are the following:

- **4.6 Software Test** gives various testing strategies to identify software errors, emphasizing the importance of test automation to ensure code quality.
- **4.7 Release Management** defines release planning, the creation of release packages (including licensing) and gives importance to automating the release process in general.
- **4.8 Automation and Dependency Management** explains the necessity of automation recurring tasks in software development, using build tools and scripts to improve efficiency and also addressing the importance of dependency management.

The scope of this thesis is limited, and does not attempt a full comprehensive evaluation of LLMs in further areas of software development. The project used for this study will be the DLR software project GitLab-Calendar[1], which provided by the DLR, serves

[1]https://github.com/DLR-SC/GitLab-Calendar

as a case study for demonstrating the feasibility and effectiveness of LLM-generated CI pipelines and software tests.

1.4. Outline

The structure of the thesis is as follows:

- **Chapter 2 - State of the Art:** An examination of existing research involving the use of LLMs within software development, with particular attention to CI/CD practices and software testing. This chapter will establish the context for the study and identify current knowledge base.
- **Chapter 3 - Methodology:** A detailed description of the methods used in this thesis.
- **Chapter 4 - Results and Analysis:** Presentation of findings of the study, including an evaluation of the LLM-generated solutions.
- **Chapter 5 - Discussion:** A reflection on the implications of the results, discussing the benefits and challenges of the work.
- **Chapter 6 - Conclusion:** Summary of key findings and contributions of this thesis.

2. State of the Art

The State of the Art chapter provides a foundation into Large Language Models and Software Engineering, while also going into detail for Continuous Integration and Software Testing, as both practices are a key aspect of this thesis. By examining existing literature, we aim to contextualize the research question of this thesis within the broader field.

2.1. Artificial Intelligence

Artificial Intelligence is a broad field of computer science with a wide range of definitions [29]. Historically, researchers have offered different interpretations, ranging from systems that think and act rationally to those that simply mimic specific parts of human intelligence [41]. In the context of this thesis, we will focus on AI as the development of computer systems that can perform tasks that typically use human intelligence, such as learning, reasoning, problem solving, perception, and language understanding [41, 11].

Artificial Intelligence has several subfields, each focusing on different aspects of creating intelligent systems [41]. Some of the major subfields include:

- Machine Learning uses algorithms that enable computers to learn from and make predictions based on data [17].
- Computer Vision deals with enabling machines to interpret and understand visual information from the world [49].
- Robotics focuses on the design and operation of robots, physically situated in the "real world", capable of performing physical tasks autonomously [29].
- Natural Language Processing is aimed to making computers understand the statements or words written in human languages [23].

From all of these subfields, NLP is particularly relevant to this thesis, as it forms the foundation of Large Language Models.

2.1.1. Natural Language Processing

Natural Language Processing emerged as an intersection between artificial intelligence and linguistics [31]. Early applications of NLP focused on tasks such as information retrieval, information extraction, automatic language translation and knowledge acquisition [11], laying the base for the sophisticated language models and applications we see today. With advancements in machine learning, particularly deep learning techniques, NLP has evolved to be able to tackle more complex language tasks like sentiment analysis, which have improved the capability of NLP systems to understand human language more effectively [19]. These advancements culminated in the development of Large Language Models.

2.1.2. Large Language Models

Language Models use statistical techniques to predict the likelihood of word sequences, generate new text based on a given input, and estimate word probabilities based on the preceding context [9]. LLMs are LM's trained on massive datasets with billions of parameters, giving them exceptional language processing abilities and strong capacities to solve tasks via text generation [9, 58]. As the foundation for Large Language Models, NLP plays an important role in advancing the capabilities of AI systems to interact with humans in more natural ways.

Fundamental to most modern LLMs, such as GPT-3 [7] / GPT-4 [34], PaLM [1] and LLaMA [51], is the transformer architecture, a neural network design introduced in the paper "Attention Is All You Need" [52]. While transformers incorporate different complex techniques, their core innovation is the self-attention module. Self-attention enables the model to weigh the importance of different words in a sentence when computing a representation, which allows complex relationships within the text to be captured. This parallelized approach improves the efficiency of processing large amounts of textual data [52].

One of the main ways in which users interact with LLMs is through conversational interfaces known as chatbots [37]. While LLMs themselves are complex models operating in the background, chatbots act as user-friendly front-ends, translating user inputs into prompts that the LLM can understand and process. The main difference between using a chatbot and sending a simple request to the LLM is that chatbots can maintain the entire conversation as context, allowing it to keep and reference information throughout the interaction [28]. This accessibility has led to a surge in the popularity and adoption of commercial LLMs across various domains. Figure 1 illustrates an interaction from the ChatGPT chatbot, showing how questions and answers are managed within the chat interface.

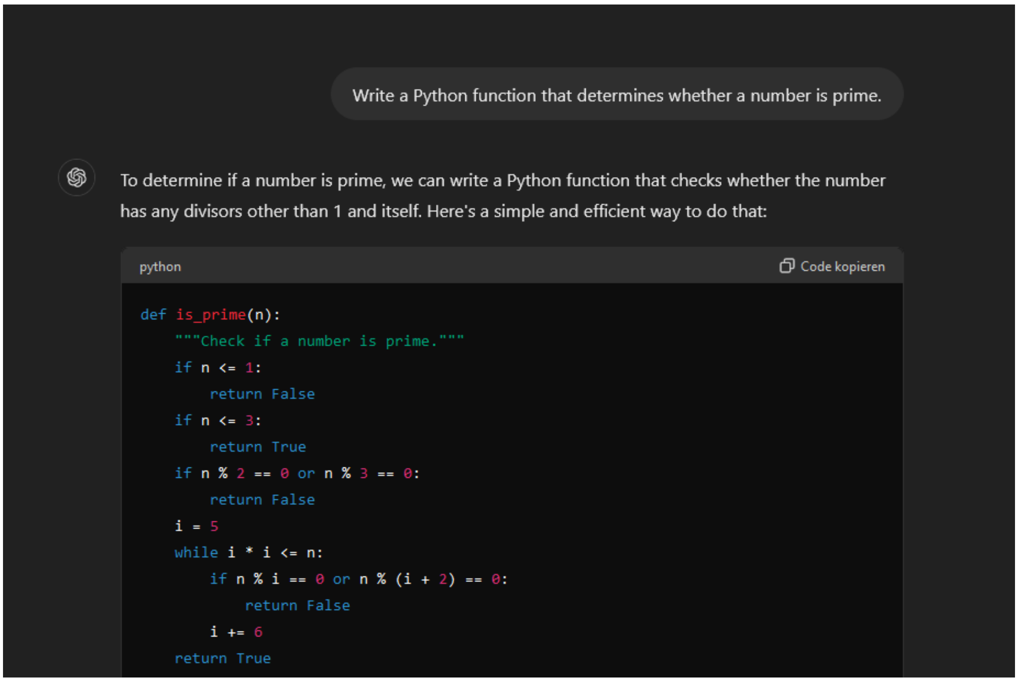

Figure 1.: Chatbot conversation for generating a is_prime function in Python.

2.1.2.1. Categories of LLMs

Large Language Models cover a wide range of purposes, with models having a variety of domains. These differences come from pre-training processes, where models are exposed to different types of text data to learn facts and reasoning abilities in a self-supervised manner [32]. Some of the most widely used LLMs include:

- General-Purpose LLMs: These models, such as GPT-3 [7] / GPT-4 [34], PaLM [1], and LLaMA [51], are designed to handle a wide range of tasks across different domains. They excel in tasks like text generation, translation, summarization, and question answering.

- Coding LLMs: Models like CodeGen [33], Codex [10], and AlphaCode [26] are specifically trained on code repositories and designed to assist with programming tasks. They can generate code snippets and debug existing code, making them useful tools for software developers.

- Scientific Knowledge LLMs: Galactica [50] is trained on scientific literature and data, enabling them to answer scientific questions, create research summaries, and assist in hypothesis formulation.

- Finance LLMs: BloombergGPT [56] is a specialized LLM trained on financial data. It is designed to perform tasks specific to the financial domain.

Figure 2 shows a chronological display of LLM releases from the paper "A Comprehensive Overview of Large Language Models" [32]. Blue cards show pre-trained models and orange cards "instruction-tuned" models. Models on the upper half have open-source availability, while the ones on the lower half are closed-source models.

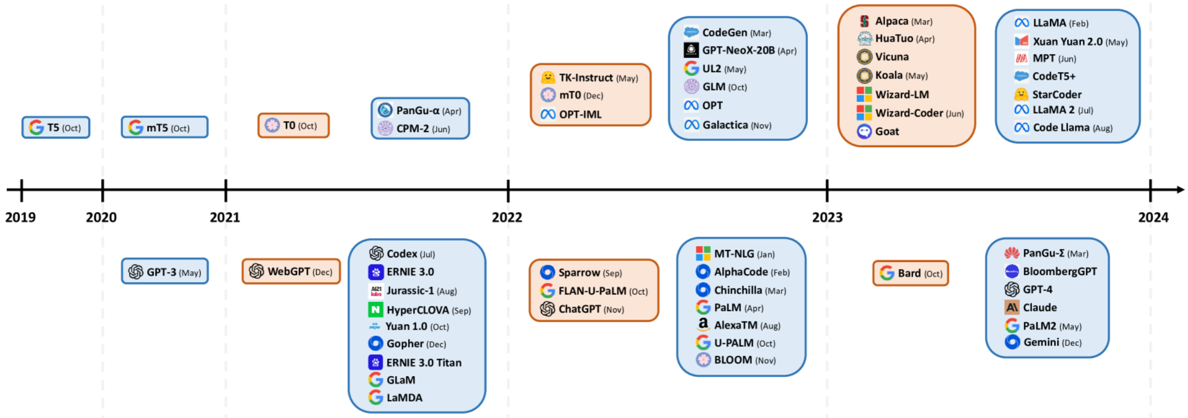

Figure 2.: Chronological display of LLM releases
Image from "A Comprehensive Overview of Large Language Models" [32]

The choice of LLM largely depends on the specific task and domain. As research in this field progresses, we can expect to see even more specialized and powerful LLMs. The paper "A Comprehensive Overview of Large Language Models" [32] by Naveed et al. offers a more extensive and in-depth examination of existing LLMs.

2.1.2.2. Prompt Engineering

Prompt engineering refers to the process of developing and optimizing input queries or prompts to use the capabilities of LLMs to produce specific, relevant, and accurate responses. This practice is necessary because responses can vary widely based on how the prompt is phrased. Effective prompt engineering can enhance the accuracy and reliability of the generated text, making LLMs more useful in practical applications. Given the potential of LLMs in various domains, prompt engineering has emerged as a skill set that significantly impacts the performance and utility of these models [53].

There are different techniques to guide LLM responses effectively. These techniques include [42]:

- Few-shot prompting provides the LLM demonstrations of the desired output to steer the model to better performance.
- Chain-of-thought prompting encourages the LLM to generate a step-by-step reasoning process before reaching a conclusion.

- Prompt Chaining involves dividing the prompt into different subtasks, making it easier for the LLM to tackle simpler tasks one by one.

2.1.2.3. Limitations of LLMs

Despite their impressive capabilities, Large Language Models face several limitations that impact their reliability and usability. One of the primary concerns is their "black box" nature, meaning it is almost impossible to understand the reasoning behind their generated responses [5]. This opacity can lead to erroneous responses, as the models may produce plausible but incorrect or nonsensical information. This is often called a hallucination. A survey performed by Rawte et al. [39] dives deeper into the different types of hallucinations and existing strategies to mitigate them.

LLMs often need enormous computational resources. Training these models demands high-performance specialized hardware, such as GPUs or TPUs. State-of-the-art models require substantial resources, which demands large amounts of energy together with the associated financial and environmental costs [48]. This resource-intensive nature limits their accessibility and scalability for many organizations on the basis of finance.

Financial cost is only one aspect of the challenges posed by the resource demands of LLMs. As mentioned before, training LLMs also has a significant environmental toll due to the high energy consumption required. The carbon footprint associated with both training and operational use of LLMs can be substantial, contributing to climate change and raising significant sustainability concerns [55]. According to Patterson et al. [36], training one large model is equivalent to 389.833 kilometers driven by an average passenger vehicle.

Another significant limitation is the inherent biases present in LLMs. These models are trained on datasets that include text from various sources, which can contain biased or prejudiced information. As a result, LLMs can reproduce and amplify these biases in their outputs, posing ethical and fairness concerns [5]. Using high-quality and diverse datasets that are representative of the real world can help fight against bias and prejudice in the outputs from the model.

2.2. Software Engineering

Software engineering is a disciplined approach to the design, development, operation, and maintenance of software. It integrates principles from computer science and engineering to create software systems that are reliable, efficient, maintainable, and scalable [46]. This section provides an overview of key concepts in software engineering, focusing on methodologies and practices used today.

As society's reliance on complex software systems continues to grow [2], software engineering has become increasingly important in recent years. By following a structured approach, software engineering helps reduce development costs, mitigate risks, and reduce errors [46].

2.2.1. Software Engineering in Research

Software engineering in research plays an important role in advancing scientific investigation and technological development. The integration of software engineering into research can lead to significant improvements in the quality and impact of scientific discoveries.

Software designed to support research activities is referred to as research software. This includes software developed specifically for scientific purposes, such as simulations, data analysis tools, and visualization applications. It can also include software prototypes in engineering research or even general infrastructure software used for research data [15].

Research Software Engineers bridge the gap between traditional research and software development. Researchers often lack a proper computer science background or the skills to develop and use specialised software for their research [54]. RSEs possess expertise in both domains, allowing them to create and maintain software that aims to answer research questions and meets the standards of scientific research [15]. RSEs often have a multifaceted role with the combination of research, software engineering, and data management. They collaborate with a diverse set of colleagues, including other developers, support staff, and academics from various fields [18].

2.2.2. Continuous Integration

Continuous Integration is a software development practice where developers regularly merge their code changes into a main repository, then perform automated builds and tests [14, 16]. This practice helps to maintain a stable codebase, reduce integration problems, and allow teams to develop cohesive software rapidly and with higher confidence. By detecting errors early in the development cycle, CI reduces the risk of bugs and enhances the overall quality of the software [14].

A typical CI pipeline consists of several key components:

- Version Control System manages and tracks changes to the source code. Tools like Git and Subversion are commonly used.
- Build Automation is the process of compiling the source code into executable files. Tools such as Maven, Gradle, and Ant are often utilized.

- Automated Testing runs unit, integration, and end-to-end tests to verify the functionality of the code.
- Continuous Integration Server orchestrates the CI process by monitoring the Version Control System for changes, triggering builds, and running tests. Popular CI servers include Jenkins, Travis CI, and GitLab CI.
- Notification System alerts developers about build and test results, facilitating a quick response to any issues. Notifications can be sent via email or in chat applications like Slack.

2.2.2.1. Relation to other methodologies

Continuous Integration is closely related to Agile and DevOps methodologies, both of which emphasize iterative development and close collaboration between team members.

Agile is a popular software engineering methodology that emphasizes iterative development, collaboration, and flexibility throughout the life-cycle of a project [12]. The Agile Manifesto [4], published in 2001 by a group of 17 software engineers, outlines a set of values and principles that guide this approach. Continuous integration is one of the core practices in agile methodologies, as it supports frequent integration and testing, allowing teams to have continuous feedback and improvement [12].

DevOps is a set of practices that combines software development and IT operations to shorten the development lifecycle and deliver high-quality software continuously [21]. The primary goals of DevOps are to improve collaboration between development and operations teams, automate processes, and enhance the efficiency and reliability of software delivery [24]. CI is a main practice in DevOps, as it ensures that code changes are regularly tested and integrated, facilitating seamless deployment processes and improved collaboration between development and operations teams.

2.2.3. Software Testing

Software testing is a critical component of the software development lifecycle, aimed at ensuring that software systems function correctly and meet specified requirements. It involves the systematic identification of defects, verification of functionality, and validation of the software's performance under various conditions [30]. Software testing is a main component of quality assurance in any software system [6]. Many studies have shown that software testing utilizes approximately 50% of the total cost of software development [38, 25].

Most of software testing can be divided into two basic categories: Manual Testing and automated Testing. Manual software testing is performed statically by a person going through the application, trying different input combinations and recording observations [43]. On the other hand automated software testing involves the development and execution of test scripts and the use of automated test tools [13]. It is a very useful approach to replace the time consuming manual testing. Automated tests can be executed quickly and repeatedly, ensuring that software updates do not introduce new defects. Automation enhances test coverage by enabling the execution of a large number of test cases across various configurations [25]. It also allows for continuous testing in CI pipelines, facilitating faster feedback and shorter development cycles [8].

2.2.3.1. Types of Software Tests

There are different ways to classify a specific test. The following test types can be distinguished [44]:

- Unit testing involves testing individual components or units of a software application to ensure they perform as expected. Developers usually write unit tests to verify that each module of the codebase operates correctly in isolation [40].
- Integration testing focuses on verifying the interactions between integrated units or components. The goal is to identify issues that arise when modules are combined, ensuring they work together. Integration tests are very important for detecting interface errors and inconsistencies between components [22].
- System testing involves evaluating the complete and integrated software system to verify that it meets the specified requirements [44]. This type of testing encompasses functional and non-functional aspects, including performance and usability [27].
- Acceptance testing is performed to determine whether the software is ready for delivery and meets the acceptance criteria set by the customers or stakeholders. It is evaluated in real world scenarios by actual users [27].

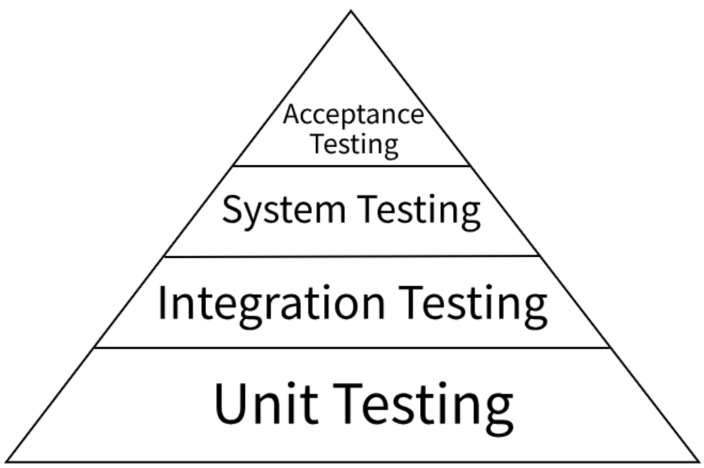

Figure 3.: Software Testing Pyramid

The distribution of these types of tests is commonly represented by the testing pyramid, as shown in Figure 3. The testing pyramid emphasizes that unit tests should form the base of the pyramid, being the most numerous. Integration tests should be fewer in number and sit above unit tests in the pyramid. System tests, which are more comprehensive and costly to run, should be fewer than integration tests and lie above them. At the top of the pyramid, acceptance tests are the least numerous but crucial for validating the software in real-world scenarios. This structure ensures a balanced approach to testing, maximizing coverage and efficiency while minimizing the cost and complexity of testing efforts.

3. Methodology

3.1. Overview

A structured methodology was used to look into how well LLMs can make CI pipelines and software tests that adhere to the DLR Software Engineering Guideline. This chapter details the selection process of the chosen LLM, the development of appropriate prompts for the specific tasks, and the project utilized in this thesis.

The approach followed in this study involves the following steps:

1. Selection of an appropriate Large Language Model based on performance and suitability for the task.
2. Selection of an appropriate software project from DLR to be the focus of this study.
3. Preparation of detailed input prompts to guide the LLM in generating CI pipelines and software tests.
    a) Summarizing the project's functionality to provide context.
    b) Writing the project structure and the most relevant files into the prompt to give an overview of the project.
    c) Selecting the relevant parts and recommendations of the Software Engineering Guideline from each section.
    d) Writing an explicit task for the LLM to perform.
4. Generation of CI pipelines and software tests using the selected LLM and prepared prompts.
5. Evaluation of the generated outputs against predefined criteria to assess their correctness, completeness, and adherence to the DLR Software Engineering Guideline.

The full conversation with the LLM, including the input prompts and responses, is documented and included in the Annex of this thesis.

3.2. Selection of Large Language Models

To conduct this thesis, we need to choose the LLMs that are most appropriate for the goal of the investigation. After careful consideration and discussion, GPT-4 [34] has been chosen as the most suitable LLM for this thesis. GPT-4 stands out as the most advanced and well-trained model available. At the time of writing, it holds the top ranking on the LLM leaderboard from LMSYS[1]. This ranking reflects the model's strong performance and reliability in various natural language processing tasks, further justifying its suitability for this study.

Although code completion LLMs such as Codex or similar models would also be suitable for generating code and configurations, the model used in this thesis must also understand and interpret the DLR Software Engineering Guideline. GPT-4 has extensive training on diverse datasets, excels not only in code generation but also in understanding complex textual information, making it a more suitable choice for this study.

Furthermore, using GPT-4 is more accessible and convenient for us, as it eliminates the need to run open-source models on a local computer or server. The ability to use GPT-4 through a simple web interface in the browser significantly reduces setup and maintenance efforts, allowing us to focus on the investigation instead on the technical overhead of managing LLM infrastructure. This ease of access and use supports the decision to utilize GPT-4 for this thesis.

3.3. GitLab-Calendar

To demonstrate the feasibility of LLM-generated CI pipelines and software tests, this study used the DLR's open-source project GitLab-Calendar[2]. This project was selected for several reasons. Its open-source availability ensures accessibility for the study. The project is simple enough to demonstrate the abilities of the LLM in a manageable context but complex enough to avoid triviality, ensuring a meaningful evaluation of the LLM's capabilities. Additionally, the GitLab-Calendar project, with its existing CI pipeline and test cases, provides a way to compare the existing solutions to those generated by the LLM, offering a useful method for assessing the results of this study.

The GitLab-Calendar project is an extension for GitLab that generates ICS files from a repository's issues, milestones, and iterations which have a due date. This functionality helps team members visualize and manage their tasks and milestones by integrating GitLab's issue tracking and milestone features into a calendar format. The

[1] https://chat.lmsys.org/?leaderboard
[2] https://github.com/DLR-SC/GitLab-Calendar

project is implemented in Python and follows standard software engineering practices.

The project originally had a CI pipeline and software tests which were not included in the prompts provided to the LLM. We will analyze the results by comparing the generated CI pipelines and software tests with the original files that were already present in the project.

3.4. Prompts

The preparation of the input prompts is an important part in using the LLM for the goal of this investigation. As a general guideline for prompt design, the guide by DAIR.AI [42] was used. This section details the steps taken to develop our prompts, providing explanations for each part of the prompt and why they were included.

To provide the LLM with the proper context needed to solve the task at hand in the best way possible, the following components were included in the prompt:

- Project Overview
- Project Structure
- Relevant Files
- Guideline Overview
- Recommendations
- Task

Each of these components plays a role in ensuring that the outputs generated are accurate and relevant. The detailed explanation of each component is provided below. The full prompt can be found in the Annex of this thesis.

3.4.1. Project Overview

The Project Overview provides the LLM with a general understanding of the project's functionality, which is especially useful when generating test cases. For the GitLab-Calendar project, the overview included a brief description of the project's purpose and main functionality, coming from the project's README[3].

> GitCalendar is a Python tool that generates an ICS file from issues, milestones and iterations, of one or more GitLab projects. Only events with a due date are considered.

[3] https://github.com/DLR-SC/GitLab-Calendar

3.4.2. Project Structure

Outlining the project structure gives the LLM insight into the organization of the project's files, which is required for creating CI configurations and relevant test cases. This is needed when referencing a specific file path, which is done in both tasks.

> - src/
>   - gitcalendar/
>   - gitcalendar.py
>
> ...

3.4.3. Relevant Files

Including files like `setup.py` and `gitcalendar.py` ensures the LLM understands dependencies, build requirements, and the logic of the application, which is necessary for generating coherent and functional CI pipelines and test cases. Some files are more important than others; for example, `gitcalendar.py` is needed for generating test cases, while `setup.py` is essential for the CI pipeline. On the other hand, files like license headers are not very important for the LLM and do not need to be explicitly included in the prompt.

3.4.4. Guideline Overview

Each section of the guideline includes an overview of one or two pages, followed by a list of recommendations with more specific requirements. The overviews are longer than needed for the prompt and could be more concise. LLMs are particularly good at summarizing information while retaining all important details. Therefore, we used GPT-4 to summarize the overview, focusing on the most relevant information while omitting less critical details.

> Section 4.8 - Automation and Dependency Management:
>
> Automation and dependency management are critical for handling software complexity, ensuring consistent builds, and reducing errors. Key components include:
>
> - Build Process Automation: Automates the transformation of source code into executable programs, including testing and release packaging. Tools like Maven and CMake facilitate this process.
>
> ...

3.4.5. Recommendations

Each section of the DLR guideline contains ten recommendations with concrete requirements for the project. However, some recommendations are out of scope for this study as they focus on more complex topics or address elements not included in our software project. The two best suited recommendations were chosen for each section. To make this selection, we prompted the LLM to identify the most suitable recommendations for each section. We then combined these suggestions with our own opinions to finalize the recommendations used in the prompt for each section. This approach ensures that the generated CI pipelines and test cases are not only functional but also adhere to established best practices relevant to our specific context.

> **Recommendations from Release Management (4.7):**
> - **ERM.6**: All foreseen test activities are executed during release performance.
> - **ERM.7**: Prior to the approval of the release, all foreseen tests passed successfully.
>
> **Recommendations from Automation and Dependency Management (4.8):**
> - **EAA.1**: The simple build process is basically automated and necessary manual steps are described. In addition, there is sufficient information available about the operational and development environment.
> - **EAA.5**: In the build process, the execution of tests, the determination of metrics, the creation of the release package, and, if necessary, other steps are performed automatically.
>
> **Recommendations from Software Test (4.6):**
> - **EST.4**: The basic functions and features of the software are tested in a near-operational environment.
> - **EST.5**: There is a test for every nontrivial error.

3.4.6. Task

Clearly stating the task instructs the LLM on what output is expected. The task description includes specific instructions on how to use the information provided and the specific goal to produce, ensuring that the generated outputs are aligned with the study's requirements.

> **Task:**
> Generate test cases and write their implementation for gitcalendar.py that adhere to the DLR guideline's recommendations provided above and the principles of test automation.

3.5. Generation of CI Pipelines and Software Tests

With the prepared input prompts, the selected LLMs were used to generate the outputs. The process involved inputting the prompts into the LLMs, and collecting and storing the generated responses.

We will analyze the results of the initial responses from the LLM without attempting to correct any possible errors in subsequent prompts. This approach allows us to evaluate the LLM's effectiveness and reliability in generating accurate and complete configurations and tests without human intervention.

By evaluating the first outputs, we can gain a clear understanding of the current capabilities of LLMs and identify areas where they excel or need improvement. Additionally, this method aligns with the objective of minimizing manual effort in the software development process, as any iterative refinement would introduce a level of human involvement contrary to the goal of automating CI pipeline and test generation.

The study aims to assist researchers who may not have extensive knowledge in software engineering. Iterating to correct mistakes made by the LLM may require a deeper understanding of the area, which may not be feasible for everyone.

4. Results and Analysis

Presented below are the results and analysis of the study, including the evaluation and examination of the generated CI pipelines and software tests. The results are evaluated based on predefined criteria, and a detailed analysis is done to understand the strengths and weaknesses of the LLM. We also provide explanations and context to improve the readability and comprehension of the findings.

The full conversation with the LLM, including the input prompts and responses, is documented and included in the Annex of this thesis.

4.1. Functionality Testing

We will execute the project for both of the tasks to evaluate whether the generated CI pipeline and software tests work as expected and integrate smoothly with the project. Running these generated outputs helps determine if they perform their intended functions correctly and if there any syntactical or logical errors that would prevent their operation. Manually reviewing the outputs without executing them, would not provide the same level of insight into real-world functionality.

4.1.1. Functionality Testing Results

Task 1: CI Pipeline

The resulting CI pipeline did not execute due to fundamental issues with the environment setup and dependencies. The first error encountered was `python: command not found`. This error occurred because the CI pipeline configuration did not specify a Docker image that includes Python. Without a Python environment, the subsequent commands in the pipeline could not be executed. The absence of a defined Python image shows an oversight in the understanding of the project, as it relies on Python for its build, test, and release processes.

Even if the Python image error had been resolved, the pipeline would have failed later due to the reference to a non-existent `requirements.txt` file. The generated configuration attempts to install dependencies using this file, which does not exist in the project. In GitLab-Calendar, the dependencies are specified in the `setup.py` file.

To improve the generated CI pipeline, it would be necessary to specify a suitable Docker image, such as `python:3.8` or any version compatible with the project requirements. Additionally, updating the configuration to correctly reference the `setup.py` file for dependency management would also be necessary.

Task 2: Software Tests

The generated tests did not execute successfully due to several errors. The first issue encountered was with the line `from gitcalendar import GitCalendar`, which incorrectly assumes that the `gitcalendar.py` file contains a class named GitCalendar. The actual `gitcalendar.py` file only defines functions and does not include any class definitions. This misunderstanding of the code provided resulted in an import error, preventing the tests from running successfully.

This outcome highlights a big limitation in the generated tests: the model's inability to accurately interpret the code provided in the prompt. Despite having access to the `gitcalendar.py` file, the generated test cases were not properly using the project's code. For the LLM to make a correct implementation of the tests, it needs to first understand the logic and functionality of the application.

4.2. Adherence to DLR Software Engineering Guideline

This section evaluates the generated CI pipeline and software tests with respect to their alignment with the DLR Software Engineering Guideline. Designed to assist researchers who may have programming skills but lack comprehensive software engineering experience, the guideline provides a structured set of recommendations aimed at ensuring correct practices in software engineering. These recommendations build upon each other across different sections, introducing principles before expanding into more specific applications for various software classes and environments. Given this approach, some recommendations appear to be similar, as they are designed to cover overlapping aspects. However, each recommendation adds specific details that are intended to guide users through increasingly more complex practices.

By comparing the generated outputs against these standards, we can evaluate the LLM's ability to produce configurations and tests that align with established quality and reliability criteria. The following sections look at how the generated CI pipeline fits with the DLR Guideline's Release Management and Automation and Dependency Management sections. They then look at how the generated software tests fit with the Software Test section.

4.2.1. Adherence to Sections Release Management and Automation and Dependency Management

Recommendations from Release Management (4.7):

> **ERM.6**: All foreseen test activities are executed during release performance.

The generated CI pipeline includes a test stage that runs prior to the release stage. This setup effectively aligns with the recommendation to execute all foreseen test activities during the release process. In a continuous integration environment, such as GitLab CI, the pipeline must successfully pass all stages, including testing, before any code can be merged or a release can be performed. This inherent characteristic of CI pipelines ensures that no untested code is released, fulfilling the intent of the recommendation.

> **ERM.7**: Prior to the approval of the release, all foreseen tests passed successfully.

The pipeline's current configuration includes a test stage that runs before the release stage. In a continuous integration pipeline, it is ensured that all tests must pass before the release stage can proceed. If any test fails, the pipeline is automatically halted, preventing the release from being approved. This mechanism aligns with the recommendation that all anticipated tests must pass successfully prior to release approval.

Recommendations from Automation and Dependency Management (4.8):

> **EAA.1**: The simple build process is basically automated and necessary manual steps are described. In addition, there is sufficient information available about the operational and development environment.

The generated CI pipeline includes a build stage, which automates the process of creating distribution packages using `setup.py`. This aligns with the recommendation for automating the build process. However, the recommendation also mentions the need for sufficient information about the operational and development environment, typically provided in project documentation, such as a README file. The CI pipeline and the documentation of a project are not related to each other. The project's documentation files are a better place to manage this information, and developers who are familiar with the project should write and maintain them. While the CI pipeline can ensure that the build process is automated, the broader documentation requirements are outside its scope and should be addressed separately.

> **EAA.5**: In the build process, the execution of tests, the determination of metrics, the creation of the release package, and, if necessary, other steps are performed automatically.

The pipeline automates several key processes, including building the project, executing tests, and creating the release package, which aligns with the guideline's emphasis on automation. The build, test, and release stages are designed to run automatically in sequence. The pipeline does not include any tools or steps for determining code quality metrics, which is mentioned in this recommendation. The LLM did not fulfill the requirement for metrics determination, likely because the prompt did not explicitly mention the need for such tools. The pipeline could benefit from incorporating tools for measuring code quality metrics in order to fully adhere to this recommendation. A more explicit prompt mentioning the inclusion of a metrics tool would have likely guided the LLM to incorporate this aspect into the CI pipeline.

4.2.2. Adherence to Section Software Test

Recommendations from Software Test (4.6):

> **EST.4**: The basic functions and features of the software are tested in a near-operational environment.

The generated tests attempt to cover basic functionalities of the GitLab-Calendar project, such as fetching issues, filtering events with due dates, and generating ICS files. However, the tests rely on mock data and do not run in a near-operational environment. The tests assume the existence of a GitCalendar class, which does not match the actual structure of the project, thereby failing to test the real functions and features in a realistic environment.

> **EST.5**: There is a test for every nontrivial error.

While the generated tests include some basic error handling, such as testing for exceptions when fetching issues, they do not provide comprehensive coverage for all nontrivial errors in the project. Important aspects, like handling missing due dates or dealing with malformed data, are not thoroughly tested. To align with this recommendation, the test suite should be expanded to cover a broader range of potential errors and edge cases within the project's functionality.

4.3. Comparison with Original Solutions

This section presents a comparison between the CI pipeline and software tests generated by the LLM and the original CI pipeline and software tests already present in the GitLab-Calendar project. We show how well the LLM can replicate project-specific needs by looking at the similarities and differences in structure and specific implementation choices. We also show where the generated solutions match or differ from the original configurations.

4.3.1. Task 1: CI Pipeline

Stages and Structure

The original CI pipeline consists of three stages: test, build, and publish. The generated pipeline also includes three stages: build, test, and release. The structure of the stages in both pipelines serves similar purposes but differs in their approach to organizing tasks. The original pipeline installs dependencies within each stage and specifies a Docker image for each step, ensuring a consistent environment across the stages. In contrast, the generated pipeline uses a `before_script` to install dependencies globally before executing any stage, which can lead to discrepancies if different stages require different environments or dependencies.

Dependency Management

The original pipeline manages dependencies using `pip install -e ".[test]"`, allowing the installation of the necessary packages directly from the `setup.py` configuration. The generated pipeline attempts to install dependencies using `pip install -r requirements.txt`, assuming the presence of a `requirements.txt` file, which is not part of the project structure. This assumption by the LLM is incorrect, as the dependency information is contained within the `setup.py` file, which was explicitly provided in the prompt.

Build and Release

In the original pipeline, the build process involves calling `python setup.py sdist bdist` to create distribution packages, followed by verification with `twine`. Additionally, the pipeline checks the installation and functionality of the package before publishing. The generated pipeline performs a similar build process using `python setup.py sdist bdist_wheel` but does not include the verification step with `twine`.

For the release process, the original pipeline uses `twine upload` to publish the package to PyPi and includes a manual approval step before publishing, enhancing control over the release. The generated pipeline attempts to upload the package to a test repository on PyPi, indicating a lack of finalization for production deployment. It does not implement a manual approval step, which could lead to premature releases.

4.3.2. Task 2: Software Tests

Coverage

While the original test cases utilize the most important functionalities of the GitLab-Calendar project by covering the core functions in the project, the test cases generated by the LLM did not understand these essential aspects. Even after explicitly including `gitcalendar.py` in the prompt, the LLM was not able to utilize the relevant functions in the tests. Instead, the generated tests included functions like `fetch_project_activities` and `parse_project_activity_data`, which do not exist in the actual project. This shows a disconnect between the generated outputs and the specific requirements and functionalities of the project.

Mocking

Both the original and generated test cases use mocking to simulate API calls and external dependencies. However, the original test cases include more complex mocking scenarios, such as mocking ProjectIssue, Project, and other project related objects, whereas the generated tests primarily mock HTTP requests.

Parametrization

The original test cases extensively use `pytest.mark.parametrize` to test multiple scenarios for functions like `merge_events` and `convert_ids`, ensuring that various edge cases and input combinations are tested. In contrast, the generated test cases do not use parametrization and test only single scenarios for each function. If the LLM had been explicitly prompted to use parametrization, it would have incorporated it in a similar manner to the original tests.

5. Discussion

5.1. Evaluation of LLM Performance

In this section, we evaluate the LLM's performance in generating CI pipelines and software tests for the GitLab-Calendar project. By examining both the strengths and weaknesses of the generated outputs, we assess the LLM's effectiveness in meeting project requirements and adhering to software engineering best practices. This evaluation considers how well the LLM's outputs function as a whole, alongside its limitations. This analysis provides insights into the LLM's current capabilities and highlights areas for improvement when applied to similar tasks in software development.

5.1.1. Strengths

The LLM demonstrated its capability to generate comprehensive CI pipelines and software tests that covered a wide range of functionalities. For example, the generated CI pipeline included stages like build, test, and release, automating critical aspects of the software development lifecycle. Similarly, the generated software tests attempted to cover various scenarios, including data handling, file generation, and error management. While not perfect, the LLM's ability to generate these outputs highlights its potential to significantly assist developers by providing templates for configuration files and boilerplate code.

One of the most notable strengths of the LLM is its ability to quickly generate configurations and test cases through writing a prompt, significantly reducing the time and effort required compared to the more time-consuming and error-prone manual process. Although the generated code required adjustments, the LLM provided a solid foundation for further refinement, significantly enhancing productivity, especially in large projects where time and resources are limited.

The LLM also demonstrated an understanding of industry best practices in the generated outputs. For example, the inclusion of `twine` in the CI pipeline reflects a well-regarded tool for managing Python package distributions. `twine` was also used in the original project. The fact that the LLM incorporated this best practice without explicit instruction suggests that it can serve as a valuable tool for enforcing standards and best practices across development teams.

5.1.2. Weaknesses

While the LLM generated functional CI pipelines and software tests, there were notable gaps in the completeness of these outputs. For instance, the CI pipeline did not include all necessary steps for creating and verifying traditional Python packages, such as the inclusion of the `twine check` command or manual verification steps that are crucial in the original pipeline. Additionally, the generated software tests lacked comprehensive coverage of non-trivial errors and edge cases, which are needed for ensuring more reliable software. These omissions show a limitation of the LLM: its inability to fully replicate the depth of understanding and attention to detail that an experienced developer might bring to the task.

The LLM made several assumptions during the generation process, some of which were incorrect and impacted the functionality of the outputs. An example is the assumption that the environment would already include Python, leading to the omission of a Docker image with Python installed in the CI pipeline configuration, which prevented the pipeline from running altogether. These incorrect assumptions about environment requirements on assumed project structures can result in non-functional outputs, requiring manual correction and intervention to align the generated code with the actual project setup. This reveals a weakness in the LLM's ability to adapt to the specific nuances of a given project.

The generated outputs did not fully align with the specific requirements of the GitLab-Calendar project. For example, the software tests assumed the presence of a class structure in the `gitcalendar.py` file that did not exist, leading to non-executable tests. These misalignments highlight the LLM's difficulty in accurately interpreting and applying project-specific requirements, leading to outputs that require further refinement and customization by developers.

5.2. Challenges Encountered

This section outlines the primary challenges encountered during this study. These challenges mainly stem from the LLM's limitations in handling environment and testing setup scenarios and the need for precise prompt engineering to guide its outputs effectively. By examining the difficulties faced in creating detailed and specific prompts, as well as the LLM's struggle with advanced configurations and dependencies, we gain insight into the areas where LLMs currently fall short. Understanding these challenges is very important for identifying strategies to improve LLM-guided development processes and achieve more accurate results.

5.2.1. Prompt Engineering

One of the significant challenges encountered during this study was the need for detailed and specific prompts to guide the LLM effectively. The specificity of the given prompts had a direct impact on the quality of the outputs. For example, when prompts lacked detailed descriptions of specific files or omitted explicit instructions regarding key requirements, the LLM often made incorrect assumptions or failed to produce fully functional outputs. This shows the importance of meticulously writing prompts to include comprehensive information about the project's structure, dependencies, and specific guidelines, ensuring that the LLM could generate outputs that closely align with the project's needs.

As this study shows, it's very hard to receive a perfect and fully functional output from the LLM after just one prompt. An iterative process would be necessary, where errors and gaps in the generated code are identified and communicated back to the LLM for correction. Initially, the LLM's outputs often contained errors or omissions that would require adjusting the prompts and re-running the generation process. While this approach can be time-consuming, it highlights the LLM's reliance on clear and detailed instructions. Each iteration would need to address specific flaws in the generated code or configurations before reevaluating the revised outputs.

5.2.2. Environment and Testing Setup

Setting up the correct environment for the generated CI pipeline posed some fundamental challenges. The LLM did not specify essential details such as the appropriate Python version or Docker image, resulting in a failed execution. These issues show that while the LLM can generate pipeline configurations, it lacks the understanding to handle essential setup tasks reliably, which are very important for smooth integration with the project's environment.

Generating tests that required complex mocking scenarios presented a significant challenge. The LLM struggled to create accurate and functional test cases that involved mocking setups, such as those required to simulate external dependencies within the project. These challenges highlighted the LLM's difficulty in handling more advanced testing scenarios, where a deep understanding of the project's architecture and dependencies is crucial for better test generation.

5.3. Reflections and Insights

This section presents reflections and insights about this study, focusing on the implications of using LLMs in software engineering. By examining the potential of LLMs to automate aspects of the development process and the critical role of human oversight, we gain a clearer understanding of the current capabilities and limitations of these models. These reflections not only highlight the benefits that LLMs offer but also emphasize the importance of expert guidance in refining and validating the generated outputs.

5.3.1. Potential of LLMs

The LLM demonstrated significant potential in automating aspects of software engineering, particularly in generating initial versions of CI pipelines and test cases. While the outputs it generated were useful, it is currently impossible to know if an LLM will ever be able to produce perfect results without human oversight. There will likely always be a need for human supervision to ensure that all requirements are met. Even though the LLM can save time by producing functional outputs quickly, it's important to acknowledge that human intervention will remain essential in verifying and refining these outputs.

5.3.2. Human Oversight

This study has reinforced the importance of human oversight in the use of LLMs for software development tasks. While the LLM was able to generate useful outputs, the need for iterative refinement and the correction of errors or assumptions illustrates that these models are not yet capable of fully autonomous software development. It is essential that the individuals overseeing the process have expertise in software engineering, as not just anyone can perform this role effectively. Developers with deep knowledge must remain actively involved, guiding the LLM with precise prompts, reviewing the outputs, and making necessary adjustments. This collaborative approach, combining the speed and efficiency of LLMs with the expertise and judgment of qualified human developers, is likely the most effective way to utilize the potential of these tools in practice.

5.4. Future Work

This section outlines potential directions for future research, trying to improving the utility and adaptability of LLMs in software engineering. Building on the insights gained from this study, future work could explore iterative refinement approaches, comparisons between LLMs, and applications across different domains in software engineering. These explorations hold the potential to advance LLM-driven software engineering, making it a more reliable and versatile asset in the future.

5.4.1. Iterative Refinement

Given the necessity of iterative refinement in achieving functional outputs, future research could explore more interactive approaches where LLMs and developers collaborate in real-time. This could involve developing tools that allow developers to interact with the LLM in a more iterative manner, receiving immediate feedback and making adjustments to prompts or code as needed. Such an approach could significantly reduce the time and effort required to produce high-quality outputs. Recent tools like FRANC [45] apply quality-aware ranking to code generated by LLMs and generate a repair prompt to further refine the code. This approach could serve as a foundation for integrating automated quality control, minimizing errors in initial outputs and enhancing reliability through automated corrections.

5.4.2. Other Large Language Models

Future research could also focus on comparing the performance of different LLMs in generating CI pipelines and software tests. While this study used GPT-4, other models may produce different results. By evaluating multiple models, future work could identify which ones are best suited for specific tasks or domains within software engineering. Different studies, such as the work by Xu et al. [57], have already done systematic evaluations of various LLMs, highlighting strengths and limitations in tasks like code generation and debugging. Such comparisons would provide valuable insights into the strengths and limitations of various LLMs and could lead to improvements in prompt engineering or model selection based on project needs.

5.4.3. Different Domains

Finally, while this study focused on a specific software project, future research could investigate the application of LLMs across a broader range of domains within software engineering. This could include exploring their use in different programming languages, development environments, or types of projects. Investigating the integration of LLMs with containerization and orchestration tools like Docker and Kubernetes, could provide valuable insights into how these models can be utilized to handle complex workflows across various stages of the development lifecycle.

6. Conclusion

The purpose of this thesis was to explore the application of LLMs in generating CI pipelines and software tests for a real-world project and testing its adherence to the DLR Software Engineering Guideline. The work investigated the following aspects:

- Evaluation of the generated CI pipelines and software tests in terms of correctness, completeness, and adherence to the guideline.
- Analysis of the potential benefits and challenges of integrating LLMs into the software development process.

The evaluation of the generated outputs, in terms of correctness and adherence to the DLR guideline, revealed both strengths and limitations. The CI pipeline that GPT-4 produced followed the DLR Guidelines closely and had a well-structured approach that was nearly identical to the structure of the original working CI pipeline in the project. However, the pipeline contained syntactic errors, such as the omission of a Python environment, leading to execution failures.

This issue reflects a tendency in LLMs to default to common configurations rather than adapting to the specifics of the project. Although GPT-4 can recognize the language of the code as Python without any difficulty, it failed to include a Python environment, likely because it defaulted to general patterns that are commonly present in its training data. This behavior shows the need for further refinement in how LLMs interpret prompts, encouraging them to focus on context-specific requirements rather than relying on typical configurations. Similarly, the generated software tests faced limitations in their failure to correctly interpret the project's structure, highlighting the importance of context in code generation.

In terms of the broader implications for software engineering, we identified potential benefits of using LLMs to automate repetitive tasks such as CI pipeline creation and test case generation. LLMs can significantly improve productivity by reducing the time developers spend on writing boilerplate code. On the other hand, this thesis also showed the challenges associated with using LLMs, including the need for precise prompt engineering and human oversight. The generated outputs required iterative refinement, and the LLM struggled with more complex or nuanced aspects of the project.

These challenges indicate that, at its current stage, LLMs are better suited for research software engineers, rather than for scientists or researchers without extensive programming experience. The need for precise prompts and adjustments makes it a tool that requires technical expertise to be fully effective, limiting its accessibility for those without a software engineering background.

Ultimately, while GPT-4 demonstrated strong potential as a tool for automating parts of the software development lifecycle, it is not yet capable of fully replacing human expertise. The LLM's outputs frequently required manual adjustments to ensure they adhered to project-specific requirements and industry standards. Going forward, the integration of LLMs into software development workflows will need to focus on improving their contextual understanding and ensuring that they can adapt to the specific requirements of a given project, while continuing to rely on the expertise of human developers to guide the outputs.

List of Acronyms

AI	Artificial Intelligence
CI	Continuous Integration
DLR	Deutsches Zentrum für Luft- und Raumfahrt
GPU	Graphics Processing Unit
LM	Language Model
LLM	Large Language Model
ML	Machine Learning
NLP	Natural Language Processing
RSE	Research Software Engineers
SE	Software Engineering
TPU	Tensor Processing Unit

List of Figures

Bibliography

[1] Anil, R., Dai, A. M., Firat, O., Johnson, M., Lepikhin, D., Passos, A., Shakeri, S., Taropa, E., Bailey, P., Chen, Z., Chu, E., Clark, J. H., et al. Palm 2 technical report, 2023.

[2] Bank, W. *World development report 2019: The changing nature of work*. The World Bank, 2018.

[3] Barenkamp, M., Rebstadt, J., and Thomas, O. Applications of ai in classical software engineering. *AI Perspectives 2*, 1 (Jul 2020), 1.

[4] Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al. Manifesto for agile software development.

[5] Bender, E. M., Gebru, T., McMillan-Major, A., and Shmitchell, S. On the dangers of stochastic parrots: Can language models be too big? . In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency* (New York, NY, USA, 2021), FAccT '21, Association for Computing Machinery, p. 610–623.

[6] Bertolino, A. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE '07)* (2007), pp. 85–103.

[7] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., et al. Language models are few-shot learners, 2020.

[8] Campos, J., Arcuri, A., Fraser, G., and Abreu, R. Continuous test generation: enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (New York, NY, USA, 2014), ASE '14, Association for Computing Machinery, p. 55–66.

[9] Chang, Y., Wang, X., Wang, J., Wu, Y., Yang, L., Zhu, K., Chen, H., Yi, X., Wang, C., Wang, Y., Ye, W., Zhang, Y., Chang, Y., Yu, P. S., Yang, Q., and Xie, X. A survey on evaluation of large language models. *ACM Trans. Intell. Syst. Technol. 15*, 3 (mar 2024).

[10] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray,

A., Puri, R., Krueger, G., et al. Evaluating large language models trained on code, 2021.

[11] Chowdhary, K., and Chowdhary, K. *Fundamentals of Artificial Intelligence.* Springer, 2020.

[12] Cohen, D., Lindvall, M., and Costa, P. An introduction to agile methods. *Adv. Comput. 62*, 03 (2004), 1–66.

[13] Dustin, E., Rashka, J., and Paul, J. *Automated software testing: introduction, management, and performance.* Addison-Wesley Professional, 1999, pp. 1–29. ISBN: 0-201-43287-0.

[14] Duvall, P. M., Matyas, S., and Glover, A. *Continuous integration: improving software quality and reducing risk.* Pearson Education, 2007, pp. 1–45. ISBN: 978-0-321-33638-5.

[15] Felderer, M., Goedicke, M., Grunske, L., Hasselbring, W., Lamprecht, A.-L., and Rumpe, B. Toward research software engineering research, June 2023.

[16] Fowler, M., and Foemmel, M. Continuous integration, 2006.

[17] Goodfellow, I., Bengio, Y., and Courville, A. *Deep learning.* MIT press, 2016.

[18] Goth, F., Alves, R., Braun, M., Castro, L. J., Chourdakis, G., Christ, S., Cohen, J., Erxleben, F., Grad, J.-N., Hagdorn, M., Hodges, T., Juckeland, G., Kempf, D., Lamprecht, A.-L., Linxweiler, J., Löffler, F., Martone, M., Schwarzmeier, M., Seibold, H., Thiele, J. P., von Waldow, H., and Wittke, S. Foundational competencies and responsibilities of a research software engineer, 2024.

[19] Gunasekaran, K. P. Exploring sentiment analysis techniques in natural language processing: A comprehensive review. *arXiv preprint arXiv:2305.14842* (2023).

[20] Hadi, M. U., tashi, q. a., Qureshi, R., Shah, A., muneer, a., Irfan, M., Zafar, A., Shaikh, M. B., Akhtar, N., Wu, J., and Mirjalili, S. Large language models: A comprehensive survey of its applications, challenges, limitations, and future prospects.

[21] Jabbari, R., bin Ali, N., Petersen, K., and Tanveer, B. What is devops? a systematic mapping study on definitions and practices. In *Proceedings of the Scientific Workshop Proceedings of XP2016* (New York, NY, USA, 2016), XP '16 Workshops, Association for Computing Machinery.

[22] Jorgensen, P. C., and Erickson, C. Object-oriented integration testing. *Communications of the ACM 37*, 9 (1994), 30–38.

[23] Khurana, D., Koli, A., Khatter, K., and Singh, S. Natural language processing: state of the art, current trends and challenges. *Multimedia Tools and Applications 82*, 3 (Jan 2023), 3713–3744.

[24] Kim, G., Humble, J., Debois, P., Willis, J., and Forsgren, N. *The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations*. IT Revolution, 2021.

[25] Kumar, D., and Mishra, K. The impacts of test automation on software's cost, quality and time to market. *Procedia Computer Science 79* (2016), 8–15. Proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016.

[26] Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with alphacode. *Science 378*, 6624 (2022), 1092–1097.

[27] Luo, L. Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA 15232*, 1-19 (2001), 19.

[28] McTear, M., and Ashurkina, M. Transforming conversational ai.

[29] Murphy, R. R. *Introduction to AI robotics*. MIT press, 2019.

[30] Myers, G. J., Sandler, C., and Badgett, T. *The art of software testing*. John Wiley & Sons, 2011, pp. 1–19.

[31] Nadkarni, P. M., Ohno-Machado, L., and Chapman, W. W. Natural language processing: an introduction. *Journal of the American Medical Informatics Association 18*, 5 (09 2011), 544–551.

[32] Naveed, H., Khan, A. U., Qiu, S., Saqib, M., Anwar, S., Usman, M., Akhtar, N., Barnes, N., and Mian, A. A comprehensive overview of large language models, 2024.

[33] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis, 2023.

[34] OpenAI, Achiam, J., Adler, S., Agarwal, S., et al. Gpt-4 technical report, 2024.

[35] Ozkaya, I. Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Software 40*, 3 (May 2023), 4–8.

[36] Patterson, D., Gonzalez, J., Le, Q., Liang, C., Munguia, L.-M., Rothchild, D., So, D., Texier, M., and Dean, J. Carbon emissions and large neural network training, 2021.

[37] Radlinski, F., and Craswell, N. A theoretical framework for conversational search. In *Proceedings of the 2017 Conference on Conference Human Information Interaction and Retrieval* (New York, NY, USA, 2017), CHIIR '17, Association for Computing Machinery, p. 117–126.

[38] Ramler, R., and Wolfmaier, K. Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 International Workshop on Automation of Software Test* (New York, NY, USA, 2006), AST '06, Association for Computing Machinery, p. 85–91.

[39] Rawte, V., Sheth, A., and Das, A. A survey of hallucination in large foundation models, 2023.

[40] Runeson, P. A survey of unit testing practices. *IEEE Software 23*, 4 (2006), 22–29.

[41] Russell, S. J., and Norvig, P. *Artificial intelligence: a modern approach.* Pearson, 2016.

[42] Saravia, E. Prompt Engineering Guide. *https://github.com/dair-ai/Prompt-Engineering-Guide* (12 2022).

[43] Sawant, A. A., Bari, P. H., and Chawan, P. Software testing techniques and strategies. *International Journal of Engineering Research and Applications (IJERA) 2*, 3 (2012), 980–986.

[44] Schlauch, T., Meinel, M., and Haupt, C. Dlr software engineering guidelines. Tech. rep., August 2018.

[45] Siddiq, M. L., Casey, B., and Santos, J. C. S. Franc: A lightweight framework for high-quality code generation, 2024.

[46] Sommerville, I. *Software engineering 9th.* Addison-Wesley, 2011, pp. 1–26. ISBN: 978-0-13-703515-1.

[47] Sridhara, G., G., R. H., and Mazumdar, S. Chatgpt: A study on its utility for ubiquitous software engineering tasks, 2023.

[48] Strubell, E., Ganesh, A., and McCallum, A. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243* (2019).

[49] Szeliski, R. *Computer vision: algorithms and applications.* Springer Nature, 2022.

[50] Taylor, R., Kardas, M., Cucurull, G., Scialom, T., Hartshorn, A., Saravia, E., Poulton, A., Kerkez, V., and Stojnic, R. Galactica: A large language model for science. *arXiv preprint arXiv:2211.09085* (2022).

[51] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., et al. Llama: Open and efficient foundation language models, 2023.

[52] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems* (2017), I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc.

[53] White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., and Schmidt, D. C. A prompt pattern catalog to enhance prompt engineering with chatgpt, 2023.

[54] Wiese, I., Polato, I., and Pinto, G. Naming the pain in developing scientific software. *IEEE Software 37*, 4 (2019), 75–82.

[55] Wu, C.-J., Raghavendra, R., Gupta, U., Acun, B., Ardalani, N., Maeng, K., Chang, G., Aga, F., Huang, J., Bai, C., Gschwind, M., Gupta, A., Ott, M., Melnikov, A., Candido, S., Brooks, D., Chauhan, G., Lee, B., Lee, H.-H., Akyildiz, B., Balandat, M., Spisak, J., Jain, R., Rabbat, M., and Hazelwood, K. Sustainable ai: Environmental implications, challenges and opportunities. In *Proceedings of Machine Learning and Systems* (2022), D. Marculescu, Y. Chi, and C. Wu, Eds., vol. 4, pp. 795–813.

[56] Wu, S., Irsoy, O., Lu, S., Dabravolski, V., Dredze, M., Gehrmann, S., Kambadur, P., Rosenberg, D., and Mann, G. Bloomberggpt: A large language model for finance. *arXiv preprint arXiv:2303.17564* (2023).

[57] Xu, F. F., Alon, U., Neubig, G., and Hellendoorn, V. J. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (New York, NY, USA, 2022), MAPS 2022, Association for Computing Machinery, p. 1–10.

[58] Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., et al. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).

Annex

A. CI Pipeline

A.0.1. Prompt:

Project Overview:
GitCalendar is a Python tool that generates an ICS file from issues, milestones and iterations, of one or more GitLab projects. Only events with a due date are considered.

Project Structure:
- src/
  - gitcalendar/
    - gitcalendar.py
    - __init__.py
- test/
  - conftest.py
  - test_gitcalendar.py
- LICENSES/
  - Apache-2.0.txt
  - LGPL-3.0-only.txt
  - MIT.txt
- setup.py

Relevant Files:
setup.py: Configuration file listing project dependencies and requirements.
<setup.py File>*
gitcalendar.py: File containing the logic of the application.
<gitcalendar.py File>*

Sections 4.7 and 4.8 Overview:
Section 4.7 - Release management:
Release management involves the distribution of stable software versions, ensuring traceability and compliance with legal requirements. Key aspects include:

- Release Planning: Determines timing and scope of releases, often supported by a ticket system to connect with change management.

- Release Process: Involves creating a development branch, applying only necessary changes, and meeting defined criteria before release. Automation of these processes helps maintain quality.
- Distribution Considerations: Ensure licensing conditions are clear and comply with export controls to avoid legal issues.

Releases must be thoroughly documented, legally compliant, and meet all quality standards before being distributed to external partners or users.

Recommendations from Section 4.7:
- **ERM.6**: All foreseen test activities are executed during release performance.
- **ERM.7**: Prior to the approval of the release, all foreseen tests passed successfully.

Section 4.8 - Automation and Dependency Management:
Automation and dependency management are critical for handling software complexity, ensuring consistent builds, and reducing errors. Key components include:

- Build Process Automation: Automates the transformation of source code into executable programs, including testing and release packaging. Tools like Maven and CMake facilitate this process.
- Build Variants:
  - Developer Build (Private Build): Conducted locally for immediate feedback on changes.
  - Integration Build: Validates changes in a neutral environment, often including additional testing steps.
  - Release Build: Produces the final distribution package, extending the integration build with additional steps for release management.

Automating these processes improves efficiency, minimizes manual errors, and ensures reproducibility of development states, supporting consistent software quality across environments.

Recommendations from Section 4.8:
- **EAA.1**: The simple build process is basically automated and necessary manual steps are described. In addition, there is sufficient information available about the operational and development environment.
- **EAA.5**: In the build process, the execution of tests, the determination of metrics, the creation of the release package and, if necessary, other steps are performed automatically.

Task:
Generate a .gitlab-ci.yml configuration file for the GitCalendar project, adhering

> to the DLR guideline's sections of automation and dependency management and the recommendations given.

* The source files were too long to include in the annex, they can be found on the github repository[1].

A.0.2. Response:

To create a .gitlab-ci.yml file for the GitCalendar project, we'll consider the recommendations for release management and automation from sections 4.7 and 4.8. The CI/CD pipeline should automate the build, test, and release processes, ensuring that all dependencies are managed, tests are run, and the final release package is created without manual intervention.

Let's examine the provided setup.py file to determine the project's dependencies and configurations, and then I'll outline the structure of the .gitlab-ci.yml file.

Step 1: Review setup.py
I'll start by examining the setup.py file to identify dependencies and any custom scripts or entry points that might be relevant to the CI/CD pipeline.

Step 2: Review gitcalendar.py
Let's take a quick look at the gitcalendar.py file to understand its entry point and how it works.

Step 3: Create .gitlab-ci.yml Based on the setup and the structure of your project, here is a basic .gitlab-ci.yml configuration that follows the DLR guidelines:

```yaml
stages:
  - build
  - test
  - release

variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"

cache:
  paths:
    - .cache/pip

before_script:
```

[1]https://github.com/DLR-SC/GitLab-Calendar

```
  - python -m pip install --upgrade pip
  - pip install -r requirements.txt

build:
  stage: build
  script:
    - echo "Building the project..."
    - python setup.py sdist bdist_wheel
  artifacts:
    paths:
      - dist/

test:
  stage: test
  script:
    - echo "Running tests..."
    - pytest test/
  dependencies:
    - build

release:
  stage: release
  script:
    - echo "Releasing the package..."
    - pip install twine
    - twine check dist/*
    - twine upload --repository-url https://test.pypi.org/legacy/
    dist/*

  only:
    - main

```

Explanation

- Stages: The pipeline consists of three stages: build, test, and release.
- Variables: We cache the pip directory to speed up builds and avoid downloading the same packages repeatedly.
- Build Stage: This stage builds the source and wheel distribution of the package using setup.py.
- Test Stage: This stage runs tests using pytest, which should be placed in the test/ directory.

- Release Stage: This stage uploads the built package to a test PyPI repository. This can be changed to the official PyPI repository when ready.
- Dependencies: The test stage depends on the build stage to ensure tests are run on the newly built package.
- Release Trigger: The release stage is triggered only on the main branch to prevent unintended releases from feature branches.

This configuration assumes you have pytest set up for your tests. You can adjust the test stage script according to your specific test setup if needed.

B. Software Tests

B.0.1. Prompt:

Project Overview:
GitCalendar is a Python tool that generates an ICS file from issues, milestones and iterations of one or more GitLab projects. Only events with a due date are considered.

Project Structure:
- src/
  - gitcalendar/
    - gitcalendar.py
    - __init__.py
- test/
  - conftest.py
  - test_gitcalendar.py
- LICENSES/
  - Apache-2.0.txt
  - LGPL-3.0-only.txt
  - MIT.txt
- gitlab-ci.yml
- setup.py

Relevant Files:
gitcalendar.py: File containing the logic of the application.
<gitcalendar.py File>*
test_gitcalendar.py: File used for Testing.
<test_gitcalendar.py File>*

Section 4.6 Overview:
Software testing is essential for detecting errors and ensuring software quality, even though it cannot guarantee correctness. Testing involves several stages:

- Module Tests (Unit/Component Tests): Verify individual components for functionality and constraints.
- Integration Tests: Focus on interactions between components to catch interface-level errors.
- System Tests: Ensure the complete software system meets specified requirements in a controlled environment.
- Acceptance Tests: Validate the software against customer requirements in the target environment.

> A robust test strategy should emphasize automation and leverage the test pyramid model, prioritizing module tests for their efficiency and ease of automation. Metrics such as test coverage provide insights into software quality and test effectiveness, helping identify trends and address issues proactively.
>
> Recommendations from Section 4.6:
> - **EST.4**: The basic functions and features of the software are tested in a near-operational environment.
> - **EST.5**: There is a test for every nontrivial error.
>
> Task:
> Write a test_gitcalendar.py file that tests the GitCalendar application with the context provided and that adheres to the DLR guideline's recommendations provided above and the principles of test automation.

* The source files were too long to include in the annex, they can be found on the github repository[2].

B.0.2. Response:

To write a `test_gitcalendar.py` file that tests the GitCalendar application according to the provided context and recommendations, we will focus on creating unit tests that cover the main functionalities of the application, including handling issues, milestones, and iterations from GitLab projects. We will ensure that our tests align with the DLR guidelines, particularly focusing on automating tests and ensuring coverage for nontrivial errors.

Let's start by examining the `gitcalendar.py` file to understand its functionality and identify key areas for testing. Then, we can draft a comprehensive test suite. Here is an overview of the `gitcalendar.py` file based on the context provided:

gitcalendar.py Overview

The file contains the logic for generating an ICS file from GitLab projects, focusing on events with a due date. Key functionalities likely include:

- Fetching issues, milestones, and iterations from GitLab projects.
- Filtering events based on due dates.
- Generating ICS files from the filtered events.

Key Areas for Testing

[2]https://github.com/DLR-SC/GitLab-Calendar

1. Fetching Data: Ensure that the application can correctly fetch issues, milestones, and iterations from GitLab.
2. Filtering Events: Verify that only events with a due date are included.
3. ICS File Generation: Test the creation of ICS files to ensure the correct format and content.
4. Error Handling: Ensure that nontrivial errors, such as network issues or missing data, are handled gracefully.

Test Strategy

1. Unit Tests: Focus on testing individual functions, particularly those involved in data fetching, filtering, and file generation.
2. Integration Tests: Verify that components interact correctly, especially in data flow from fetching to ICS generation.
3. Edge Cases: Test with a variety of input scenarios, including empty datasets, malformed data, and API errors.

Now, let's draft the `test_gitcalendar.py` file based on this strategy. I'll review the contents of the uploaded files to align the tests with the actual implementation.

```
# test_gitcalendar.py
import pytest
from unittest.mock import patch, MagicMock
from gitcalendar import GitCalendar

# Mock data for GitLab issues, milestones, and iterations
mock_issues = [
    {'id': 1, 'title': 'Issue 1', 'due_date': '2024-08-01'},
    {'id': 2, 'title': 'Issue 2', 'due_date': None},
]

mock_milestones = [
    {'id': 1, 'title': 'Milestone 1', 'due_date': '2024-08-10'},
    {'id': 2, 'title': 'Milestone 2', 'due_date': None},
]

mock_iterations = [
    {'id': 1, 'title': 'Iteration 1', 'due_date': '2024-08-15'},
    {'id': 2, 'title': 'Iteration 2', 'due_date': None},
]

```

```
@pytest.fixture
def git_calendar():
    return GitCalendar(api_token='fake_token', project_id=12345)

def test_fetch_issues(git_calendar):
    with patch('gitcalendar.GitCalendar.fetch_issues',
    return_value=mock_issues) as mock_fetch:
        issues = git_calendar.fetch_issues()
        assert len(issues) == 2
        mock_fetch.assert_called_once()

def test_filter_events_with_due_date(git_calendar):
    filtered_issues =
    git_calendar.filter_events_with_due_date(mock_issues)
    assert len(filtered_issues) == 1
    assert filtered_issues[0]['title'] == 'Issue 1'

def test_generate_ics_file(git_calendar):
    filtered_events =
    git_calendar.filter_events_with_due_date(mock_issues +
    mock_milestones + mock_iterations)
    ics_content = git_calendar.generate_ics_file(filtered_events)
    assert 'BEGIN:VCALENDAR' in ics_content
    assert 'BEGIN:VEVENT' in ics_content
    assert 'SUMMARY:Issue 1' in ics_content
    assert 'SUMMARY:Milestone 1' in ics_content
    assert 'SUMMARY:Iteration 1' in ics_content

def test_handle_errors(git_calendar):
    with patch('gitcalendar.GitCalendar.fetch_issues',
    side_effect=Exception('API Error')):
        with pytest.raises(Exception) as excinfo:
            git_calendar.fetch_issues()
        assert 'API Error' in str(excinfo.value)

# Additional tests can be added for milestones and
# iterations similarly
```

Explanation

- Mocking: We use `unittest.mock.patch` to mock API calls and control the data returned for testing.
- Fixture: A fixture (git_calendar) is used to create an instance of GitCalendar

for use in tests.

- Test Cases:
  - `test_fetch_issues`: Tests the ability to fetch issues from GitLab.
  - `test_filter_events_with_due_date`: Verifies that only events with a due date are considered.
  - `test_generate_ics_file`: Tests the generation of an ICS file from events.
  - `test_handle_errors`: Ensures that exceptions are handled appropriately.

These tests cover the basic functionality and ensure robustness against nontrivial errors. We can further extend these tests with more specific cases or additional features as needed.