MASTER'S THESIS 2024

Generating Test Cases Using Natural Language Processing

Yamen Albdeiwi, Mohammed El-Khalil

ISSN 1650-2884
LU-CS-EX: 2023-79

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-79

Generating Test Cases Using Natural Language Processing

Testfallsgenerering med hjälp av Natural Language Processing

Yamen Albdeiwi, Mohammed El-Khalil

Generating Test Cases Using Natural Language Processing

(Evaluating Large Language Models for Automated Test Case Generation)

Yamen Albdeiwi
mo4718al-s@student.lu.se

Mohammed El-Khalil
mo4382el-s@student.lu.se

February 1, 2024

Master's thesis work carried out at Axis Communications AB.

Supervisors: Pierre Nugues, pierre.nugues@cs.lth.se

Examiner: Jacek Malek, jacek.malec@cs.lth.se

Abstract

Software testing today is a vital factor in maintaining the quality and reliability of software products in an always-advancing technical era. Nevertheless, making manual high-quality test documents has been historically laborious and lengthy, which results in significant time and money consumption for the whole software creation process. Recently, the use of transformer models has become an efficient tool for the automation of this procedure. This thesis is based on the use of large language models to create test case documents from feature specifications written in natural language.

We assess different ways to improve our model's effectiveness, such as fine-tuning, prompt engineering, and agentic workflow methods. We carry out the research with the use of a quantized and optimized model for memory efficiency that demonstrates the possibility of generating good test cases even with the restriction of computational resources. With our approach, we achieved remarkable results in both the BLEU score and the human evaluation score. Our highest **BLEU** score achieved with our best approach is **32.93**, which also corresponded to the highest human evaluation score **3.71**. This is not far from the reference value **4.68** of being humanly written and undergoing a review process.

Keywords: NLP, Test cases, Feature specification, Prompt engineering, Fine-tuning, Transformers, Large Language Models, Mistral 7B, Hyperparameter optimization, text2text

Acknowledgements

To begin with, we would like to thank our supervisor from LTH, Pierre Nugues, sincerely, for his invaluable guidance, unshakable support, and astounding professionalism which hugely influenced the development of this Master's thesis. Pierre's extensive expertise in the research field served us both as a guiding force to navigate through the complexities of the research process and as a source of light to highlight the very themes we aimed to cover and open new research paths.

Also, our great appreciation goes to Ingemar Larsson, AI lead within our department at Axis Communication, who shared the idea of the agentic workflow method as an applicable strategy for our case. Gradually, Ingemar's advice led to a completely new way of exploration that enabled us to make our assessment and verification of the method's efficiency and applicability according to our purposes.

Contents

1 Introduction **7**
- 1.1 Research Expectations . . . 8
- 1.2 Scientific contributions . . . 9
- 1.3 Division of Work . . . 9

2 Background **11**
- 2.1 Software testing . . . 11
  - 2.1.1 Testing optimization . . . 12
  - 2.1.2 Test Case Documents . . . 12
- 2.2 Natural Language Processing . . . 13
- 2.3 Hyperparameter Optimization (HPO) . . . 14

3 Related work **17**
- 3.1 Generation of Test Cases . . . 17
- 3.2 Large Language Models and Generation . . . 19

4 Data **21**
- 4.1 Data Collecting . . . 21
  - 4.1.1 Phase I . . . 22
  - 4.1.2 Phase II . . . 22
  - 4.1.3 Phase III . . . 23
- 4.2 Exploratory Data Analysis . . . 23
  - 4.2.1 Phase I . . . 24
  - 4.2.2 Phase II . . . 29
  - 4.2.3 Phase III . . . 32
  - 4.2.4 Employed Test Cases . . . 35
- 4.3 Data Augmentation (DA) . . . 35
- 4.4 Dataset . . . 37
  - 4.4.1 Real Data . . . 37
  - 4.4.2 Real & Augmented Data . . . 37

5 Architecture **39**
5.1 Text Processing and Generation . . . 39
5.1.1 Numerical representation of Text . . . 39
5.1.2 Text Generation and Language Modeling (LM) . . . 41
5.2 Transformer's Architecture . . . 42
5.2.1 Attention Mechanism . . . 42
5.2.2 Encoder-Decoder . . . 43
5.3 Large Language Models (LLMs) . . . 45
5.3.1 Pre-trained Language Models (PLMs) . . . 45
5.3.2 PLMs Architecture . . . 46
5.3.3 Fine-tuning Large Language Models . . . 48
5.3.4 Prompt Engineering . . . 51
5.3.5 Mistral-7B . . . 52
5.4 Llama-cpp-python Integration . . . 58

6 Experiments **61**
6.1 Overview of the Approach . . . 61
6.2 Setup & Deployment . . . 61
6.2.1 Choice of Model . . . 62
6.2.2 Defining Setup . . . 62
6.3 Running Experiments Strategies . . . 65
6.3.1 Prompt engineering approach . . . 65
6.3.2 Fine-tuning approach . . . 69
6.3.3 AI Agentic Workflow Approach . . . 70
6.4 Evaluation . . . 72
6.4.1 NLG Evaluation Metrics . . . 72
6.4.2 Human Evaluation . . . 73
6.4.3 Evaluation Pipeline . . . 74

7 Results & Discussion **77**
7.1 Fine-tuning Results . . . 77
7.2 Prompt Engineering Results . . . 78
7.2.1 Base Model with QLoRA . . . 78
7.2.2 Model with GGUF through Llama-cpp-python . . . 79
7.3 Agentic Workflow Results . . . 80
7.4 Choosing a Final Model . . . 81
7.5 Building a Practical Tool . . . 85

8 Conclusion **87**

References **89**

Appendix A LLM Evaluation and Hyperparameter Optimization Pipeline **101**

Appendix B Experiments Tracking **103**

Chapter 1
Introduction

Software testing is a vital part of the software development life cycle. Almost every aspect of modern life has some kind of integrated technology attached to it, either as an assisting tool or a critical component of that aspect. Therefore, the software offered today to end-users must be quality-assured in functionality, reliability, performance, and safety.

In the field of software testing, the research shows that software testing is the most costly segment in Software development, and it could be as high as 52% of the overall software expense. The fact that large amounts are spent on obtaining quality-assured software products shows the significance of testing processes.

One of the key elements of a reliable software testing process is good-quality test case documents. These documents are complete and informative to assess the functionality and performance of the software since they define the criteria for evaluating the performance of the software based on the environment where the software is being run, how the software runs, and what the expected results are. Thus, by following the instructions laid out in the testing framework, the results can be interpreted to assess the accuracy of the software's behavior. Nevertheless, currently, the industry standard is to write these test case documents by hand, which has been for a long time considered a time-consuming and labor consumption process. Testing engineers need enough time to scrutinize feature or software specification documents in order to generate test cases that are both related and ensure optimal testing coverage. This process is by nature susceptible to human error and lack of uniformity.

Recent advancements in the natural language processing domain have resulted in the emergence of new, innovative solutions for automatizing sub-processes within the software testing pipeline. Specifically, the development of neural network models with transformer-based architectures has contributed to models being proficient in tasks like text generation and language translation. This is due to an extraordinary increase in natural language understanding and processing capabilities.

The arising of transformer-based models has introduced extremely efficient techniques for the automation of test case generation, which was previously impossible. In this Master's thesis, motivated by the advancements of these models with their exceptional contextual

understanding and generative capabilities, we applied large language models to generate test case documents from written descriptions.

We investigated many diverse approaches aimed at improving our test case generation model capabilities. Particularly, we carried out fine-tuning techniques, including and without hyperparameter optimization utilizing supervised fine-tuning training to enrich the model's comprehension of the testing arena. Furthermore, we tried prompt engineering techniques, namely zero-shot, one-shot, and multi-shot, used to guide the model's generation competently. Another aspect we assessed was the effectiveness of a combination of fine-tuning and prompt engineering, identifying where these strategies could be used together to boost both the quality and relevance of the generated test cases. As a part of our investigation, we aimed to attain the best computational effectiveness with no compromise on the performance. Hence, all our investigations were based on the quantized model, which was optimized for memory efficiency, and was run on a single local consumer GPU, making our method possible even in real-world situations with rather limited computational capabilities.

We carried out our research at Axis Communications, where our objective was to explore the feasibility of generating test case documents directly from feature and specification documents using state-of-the-art models.

At the end of our research, we concluded that optimization of smaller large language models with 7 billion parameters is possible even with limited resources of computational power. This has been demonstrated by creating quality test cases based on requirements documents. We also determined that a lot of data is needed to fine-tune the language models to increase the performance and generation quality of the model. The results of our study have proven that when the data is not sufficient then prompt engineering is the best optimization method. These findings both contribute to the knowledge base of language model optimization and provide useful solutions to the challenges of model optimization with limited resources.

1.1 Research Expectations

The objective of this thesis is to explore the feasibility and effectiveness of generating test cases using Natural Language Processing (NLP). Our investigation focuses on the following questions:

- Can a small-sized large language model (7 billion parameters) with quantization be effectively applied in the software domain using consumer-grade GPU hardware?
- Is it feasible to generate test cases based on feature descriptions using a small-sized large language model?
- How does this approach compare in effectiveness to human-written test cases that have undergone planning and reviewing phases?

1.2 Scientific contributions

This thesis offers several interesting insights into applying NLP to software testing. First, it shows the capability of a relatively small-sized language model (7 billion parameters) with quantization in the software domain with the use of consumer-grade GPU hardware. Second, it demonstrates the usefulness of a language model for generating test cases from feature descriptions as an alternative method for automating this aspect of software testing. It also compares the results of the test cases generated by NLP with the traditional human-written test cases planned and reviewed, with such strengths and limitations clearly described.

The following outcomes helped to describe what is the most appropriate way to work with the stated situations. Limited computational resources and data made fine-tuning the worst possible option, as it only degraded the pre-trained weights. On the other hand, it turned out that prompt engineering was the best choice. Our study revealed the type of prompt engineering that best suits our task and generated the best performance both in standard metrics and human evaluation metrics.

Through these contributions, this thesis contributes to the advancement of the application of NLP techniques in the software testing area.

1.3 Division of Work

Both authors have contributed equally to both the experimentation and writing aspects of this thesis. The division of responsibilities for specific topics is outlined in Table 1.1.

Table 1.1: Division of responsibilities between the authors

Technical Topics	**Yamen Albdeiwi**	**Mohammed El-Khalil**
Experiment design and setup	50%	50%
Data collection and preprocessing	50%	50%
Exploratory data analysis	70%	30%
Implementation	50%	50%
Fine-tuning experiment	40%	60%
Prompt engineering experiment	40%	60%
AI agentic workflow experiment	80%	20%
Hyperparameter Optimization	50%	50%
Evaluation and analysis	50%	50%
Results interpretation	50%	50%
Implementing the final tool	20%	80%
Report Topics	**Yamen Albdeiwi**	**Mohammed El-Khalil**
Writing of the thesis	50%	50%
Literature study & references	50%	50%
Graphic design	50%	50%

Chapter 2
Background

This chapter provides essential context and background information for the research we have undertaken. The first section delves into the domain of software testing (ST), explaining its fundamental concepts, methodologies, and challenges associated with guaranteeing sufficient software quality. In the realm of software testing, the section also covers testing optimization and its different techniques for increasing efficiency and effectiveness in the ST cycle. Transitioning from ST, the second section dives into the field of Natural Language Processing (NLP), providing a comprehensive overview of the subject and demonstrating its application in today's digital world.

2.1 Software testing

According to Whittaker (2000), ST is the execution process of a software system with the main objective of assessing if the software executes as outlined in its specifications and works in its designated environment. Further, Singh and Singh (2012) state that the following five elements: test strategy, testing plan, test cases, test data, and test environment are the essential components to ensure a successful ST process.

Generally, it is accepted that it is impossible to achieve a perfect software or system without any flaws. Therefore, it is necessary to apply ST to a software or system before its release and throughout its whole development life cycle. However, it is equally important to perform and have high testing standards continuously. This is to decrease the risk of having a detrimental effect when the software is being used by end-users (ISO/IEC/IEEE International Standard, 2013). ST is resource and time-consuming, which leads to one of the primary challenges in the software industry. Regarding the costs, estimates have shown that up to 80% of a software development project's total cost is spent on testing tasks (Alaqail and Ahmed, 2018).

2.1.1 Testing optimization

As aforementioned, software testing is a crucial aspect for tech organizations developing software, as it provides quality assurance and also accounts for a significant part of its expenses as it amounts to more than 52% of the cost of a software development life cycle (Kiran et al., 2019). This is because a testing process is time-consuming, labor-intensive, and monotonous (Kiran et al., 2019). The factors mentioned above are the most significant driving factors for the development to improve the efficiency of the ST domain.

In recent years, testing optimization has emerged as a popular research topic in the realm of ST, which has resulted in researchers introducing several optimized testing techniques that are more efficient in terms of both labor and computation. ST is a collective term that considers many types of testing areas and techniques, where the most common testing subdomains are *test case generation*, *test case selection*, and *test case prioritization* (Gupta et al., 2019).

Test case generation: Test case generation is one of the critical activities in software testing, and it is a task that demands a lot of work and attention. It is, therefore, quite a complex task, and it influences the efficiency and effectiveness of the whole testing process. According to this, this task has been the subject of much research in the area of software testing for many years. Several methods and tools have been devised or invented to solve this task in a better way. In this aspect, efficiency refers to having as few test cases as possible but with the highest possible test coverage with as few human reviewers included in the manual work of evaluating the coverage of the test cases (Anand et al., 2013). The most prominent way of optimizing the test case generation process is to automatize the process in a fashion that improves the test coverage or at least does not inhibit it.

2.1.2 Test Case Documents

Test case documents are detailed descriptions of how software should be tested. Usually, they are written in plain natural language, making them easy to understand. The documents mainly cover topics such as the test objectives, the criteria for a successful test, and how the tests should be executed, see Figure 2.1. Depending on the scenario, more details like code blocks, images, and extra topics can be included in the testing documents.

Potuzak and Lipka (2023) explain that in the ST domain, testing documents play a significant role since they give clear direction and instruction to testers about objectives, criteria, input data, expected outputs, and testing environments for consistency and comprehension. These testing documents identify issues at an early stage and lower defects to mitigate risks and improve overall quality. They make the testing processes efficient, thus saving time and effort, as well as allowing automated tools to produce test scripts quickly. Besides tracking testing activities, testing documents serve as important documentation for traceability, which in turn helps in delivering high-quality software products.

In different stages in a software development life cycle, these documents could be found useful in different aspects. They can be used from the beginning as guidelines for deciding the functionality of software being developed until the end of when the software is being tested. Furthermore, the testing document is not only useful for developers and testers. They are also used as inputs to automated testing software to optimize the testing process by making

it more accurate and efficient by saving time and human resources (Aoyama et al., 2021). Additionally, the documents can be used as data samples to use as references when generating new testing documents, either by humans or machines.

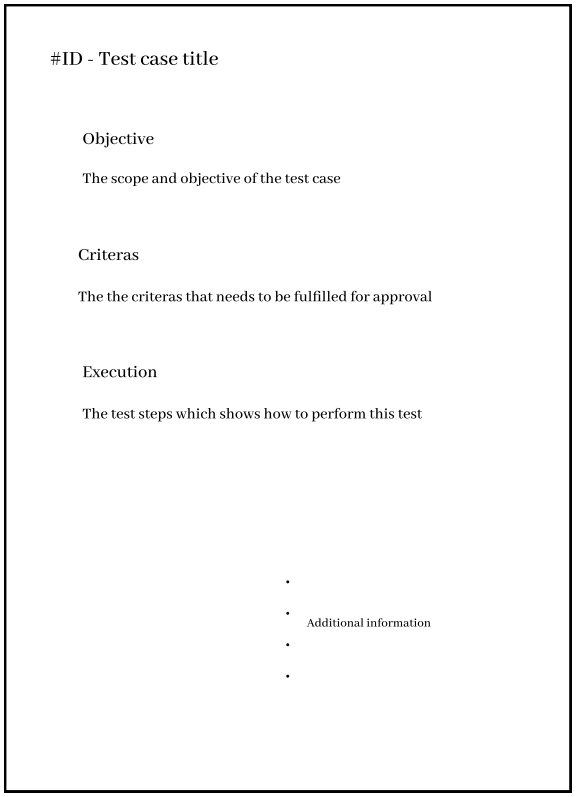

Figure 2.1: A mock of a testing document

2.2 Natural Language Processing

In the 1950s, NLP came into existence at the intersection of artificial intelligence and linguistics. Initially, NLP and text information retrieval (IR) were separate disciplines. The latter depends on statistical methods to index and search vast amounts of documents quickly. However, with time, NLP and IR have started to get intertwined more and more.

Nowadays, NLP is a multidisciplinary field with a lot of unique tasks to be solved, and researchers and developers sometimes have to widen their experience significantly (Nadkarni et al., 2011). Among IR are text classification, text generation, language translation, and text sentiment analysis key subjects in the NLP realm. Developers attempt to utilize machine learning, linear algebra (vector mathematics), and information theory to address most NLP tasks. Today's capacity of storing natural language samples such as books, articles, and transcriptions digitally favors NLP advancements significantly. Proof of this is the wide application of NLP techniques in recent years in most everyday applications, for example, spelling corrections in standard mobile phones, machine translation engines like Google Translate, speech engines like Apple's Siri, and today's mass development of interactive virtual agents like chatbots (Ferrario and Nägelin, 2020).

In the preceding years, NLP has had major advancements that have led to transitions in text processing approaches which for simplification purposes Ferrario and Nägelin (2020) divide into three categories: The *classical approach*, *modern approach*, *contemporary approach*.

Classical Approach. The classical approach was based on the generation of bag-of-words and bag-of-POS (parts of speech) representations of a text or document. The bag-of-words representation reflected the number of appearances each word has in the text, while the bag-of-POS represented the frequency of each part of speech in the text. These representations were in numerical vectors which were fed into the NLP model and based on these would the model make predictions (Ferrario and Nägelin, 2020).

Modern Approach. Diverting the discrete numerical vectors used in the previously outlined method, the modern approach aims to utilize embedding algorithms that generate representations with continuous numerical vectors. Being continuous results in significantly more mappings between word and numerical representations. This yields an improved capability of capturing the contextual meaning of the words and encompassing long-range dependencies between the words. Since the numerical vectors encapsulate contextual meaning, the model can generate more accurate predictions (Ferrario and Nägelin, 2020).

Contemporary Approach. With the advancements done in the past decade in the field of NLP, a large part of text preprocessing has been eliminated. This is because today, one can use and train neural networks directly on the text (Ferrario and Nägelin, 2020).

This contemporary approach is extremely promising, representing a significant advancement in the field. In the rest of this thesis, we will explore the challenges of how this approach can be effectively used in the automated generation of test cases from feature specifications. Thus, in this paper, we sample various methodologies and techniques, and we aim to showcase the practical applicability and potential impact of this approach in the field of software testing.

2.3 Hyperparameter Optimization (HPO)

Since the aforementioned PEFT methods overcome the need for a large amount of computer power, but leave a big selection of model parameters untouched, such methods are sensitive to the choice of hyperparameters. However, human-based selection of hyperparameters is a time-consuming task, but it can dramatically improve model performance (Tribes et al., 2024). As Bergstra et al. (2011a) discussed, finding better hyperparameters should enhance the performance, therefore HPO is considered a crucial step in bringing some efficiency.

Optimization techniques are common strategies applied to improve the performance of machine learning models. They aim to minimize or maximize a predefined metric during training, thus enhancing the model's ability to generalize and perform well on unseen data.

Early stopping algorithms are used during optimization to stop a trial, which is predicted to have poor performance (produce a suboptimal final result) since running the trial until it finishes will be inefficient.

Neural Network Intelligence (NNI) is an open-source automated machine learning (AutoML) toolkit that can automate HPO, dispatch, and run experiments' trial jobs generated by tuning algorithms to search the best hyperparameters (Microsoft, 2021). NNI has various built-in optimization estimators which are called tuners, and two built-in early stopping algorithms which are called assessors.

In this thesis, we initially examined two optimization techniques, *Random Search* and *Tree-structured Parzen Estimator (TPE)* (Bergstra et al., 2011b). Furthermore, random search is based on randomly selecting hyperparameter values from predefined ranges, but it does not benefit from any information gained during the optimization process, thus it may require a vast number of trials to find the optimal values.

On the other hand, TPE is based on the Bayesian technique building a probability distribution of promising hyperparameter values based on the performance of previous trials. Thereby, it selects the next values by balancing exploration and exploitation.

Moreover, as Bergstra et al. (2011b) discussed, TPE models $P(x|y)$ and $P(y)$ where x represents hyperparameters and y is the evaluation result. It defines $P(x|y)$ using two densities as follows:

$$p(x|y) = \begin{cases} l(x) & \text{if } y < y^* \\ g(x) & \text{if } y \geq y^*, \end{cases}$$

where $l(x)$ is the density formed by using the observations such that the corresponding loss was less than y^*, and $g(x)$ is the density formed by using the remaining observations.

In contrast, for the assessors, we explored the two supported by NNI, namely *Median Stopping Rule* and *Curve Fitting Rule* (Microsoft, 2021). The median-stopping rule stops the optimization for a trial if the value of the running trial is less than the median value of the running averages of all completed trials. Alternatively, the curve-fitting rule peeks at the performance curve, and given a set of executed trials, it performs regression on the curve to predict the final objective value of the current trial. Thereafter, if the probability of overstepping the optimal value is found low, then it stops the optimization (Golovin et al., 2017).

Thus, we chose TPE as our optimization technique since it is a better approach. Since it focuses on promising regions of the search space and, subsequently, converges to better outcomes with fewer evaluations than random search. Additionally, we decided to use the median-stopping rule as our assessor.

Chapter 3
Related work

This chapter will provide an in-depth analysis of research efforts done in test case generation, as well as using Large Language Models (LLMs) for text generation. This is a way of gaining insight into the current state of this research, thereby providing the essential groundwork for our investigation. Namely, it focuses on the details of test case generation and the use of LLMs for test case generation. It is our intention to explore these topics in more depth so that the strengths and weaknesses of the current domain will be clearly outlined.

3.1 Generation of Test Cases

Previous research has explored the capability of generating test cases using NLP.

Salman (2020) delved into the exploration of generating test cases from specification documents using NLP. In his investigation, he utilized the NLP methodologies of K-nearest neighbor (KNN) and linear support vector (LinearSVC) classifiers. Salman's approach involved parsing and analyzing test case specifications in natural language to generate feature vectors that later were mapped to test scripts. The mapping was further used as input and output in a multi-label text classification process using the aforementioned classifiers. The study showed that LinearSVC in combination with data augmentation was the optimal technique which yielded the best results, as high as 89% F1 score.

Gupta (2023) investigated the possibility of improving the efficiency of test automation processes by applying NLP techniques. Gupta used test specification documents as his data source. He extracted keywords from these documents by utilizing NLP techniques to use as input data, and the output would be the document from which the keywords were extracted. Further, he deployed algorithms including naive Bayes, K-nearest neighbor, linear support vector machine, and Random Forest where the Random Forest algorithm yielded the highest accuracy of 94%. The study proved a higher efficiency in the test automation processes and demonstrated the beneficial role of NLP in this domain.

Wang et al. (2020) provided an innovative system-level automatic test case generation solution using NLP. In their work, they adopt *Use Case Modeling for System-level, Acceptance Tests Generation* (UMTG), an approach that generates executable system test cases for acceptance testing by leveraging the behavioral information in use case (UC) specifications written in natural language. UMTG was based on the state-of-the-art algorithms at that time in natural language processing to automatically identify test scenarios and generate formal constraints that described conditions that led to the execution of these scenarios. This work utilizes NLP to build Use Case Test Models from UC specifications, which lack an explicit control flow and enable the model-based identification of UC scenarios, which represent sequences of steps within the model. According to two industrial case studies, the ability of UMTG is substantiated by translating 95 percent of the use case specification steps into formal constraints for test data generation. Additionally, UMTG created an additional set of test cases that cover the test scenarios experts applied manually as well as incorporate these critical scenarios.

Ansari et al. (2017) endorse an automated approach for producing test cases from behavioral specs by means of Natural Language Processing (NLP). Their system eases the process by pulling out specific information from Software Requirement Specification (SRS) documents and converting it into test cases. Their approach involves three main steps: input, processing, and output. Next, the functional requirement documents will serve as input, which will be analyzed through NLP in order to identify the critical scenarios and situations. Test cases are built in the context of the scenarios and conjunctive statements. The nouns are used as test data, the 'if' statements are used as test steps, and the 'then' statement is used as the result expected. An algorithm describes the iterative process of analyzing the documents, separating useful data from noise, and thus generating test cases. They demonstrate in their paper how the applicability of the system could be benchmarked against the randomly generated test cases for different scenarios. This proved that their proposed system is able to convert requirements into working tests.

Lafi et al. (2021) introduced an elegant approach for generating test cases from use case descriptions. Their suggested solution uses Unified Modeling Language (UML) which is the structured modeling language, consisting of interrelated diagrams. UML helps developers in the process of defining, visualizing, constructing, and documenting the necessary elements of software systems, such as objects, and non-software systems like business processes. It starts with taking textual information from UML use case descriptions as the basic input for the method. The data gathered then is applied to create a control flow graph and a Natural Language Processing (NLP) table, which are the critical elements needed to know the system and its dependencies. With certain specialized algorithms, the control flow graph would then be generated to illustrate different execution paths in the software. The nodes in the graph represent the fundamental building blocks of code, whereas the arcs indicate the way the code flows between these blocks. Parallel to this, the NLP table assigns a node in the flow chart to each action in the use case scenario. Afterward, the method applies yet another algorithm to produce detailed test scenarios aimed at checking the system's functional faultiness completely. Ultimately, the test paths and NLP table are transformed into automated test cases. Every test case is assigned a route and acts as a sequence of steps recorded from the control flow graph and the NLP table.

3.2 Large Language Models and Generation

In the past years, the realm of NLP has taken a huge leap, propelled by the introduction of transformers presented in Vaswani et al. (2017). Transformers paved the way for the development of LLMs, which redefined the landscape of NLP. LLMs are today's state-of-the-art NLP solutions. They are built on the aforementioned deep learning network architecture concept named transformers, which will be discussed more deeply in the architecture chapter 5. LLM models vary a lot in size depending on their complexity, but they are usually considerably larger compared to model standards previous to LLMs. For example, OpenAI's famously known GPT3 model has 175 billion parameters and consists of 96 layers, which is far larger than anything we have seen before (Chauhan, 2022).

LLMs possess extraordinary abilities to recognize contexts and capture long-range dependencies between words due to their architecture, and therefore have very broad applicability. One of the most common applications is text generation, which is what we will be utilizing in our work. When an LLM is trained on vast data, it is evaluated on its capacity to forecast the subsequent words or tokens within a provided sequence by analyzing the given sequence and then applying various statistical and probabilistic techniques (Zhang et al., 2024).

In our literature study, to the best of our efforts and knowledge, we did not identify studies exploring the possibility of generating test cases in natural language using LLMs.

Wang et al. (2024) analyzed 102 studies that were found relevant to the field of using LLMs for software testing. The undertaken investigation was conducted from the perspectives of LLMs and software testing to lay a road map for future research in that area. The study also summarizes and highlights the key challenges and potential opportunities for future work. The research resulted in a comprehensive review of using LLMs in software testing, which explains that LLMs have proven to be successful when applied in the software testing domain. The LLMs were mainly incorporated in a testing task through fine-tuning of the model or prompt engineering. Further, it mentions that the primary challenge today lies in obtaining high testing coverage and that LLMs are only applied to a subset of an entire testing cycle.

Nevertheless, None of these studies match exactly our objectives. However, as shown in other experiments, LLMs produced extremely valuable results in fields related to our task, such as *test case selection from test specifications*, where Ayenew and Wagaw (2024) adopted an AI-enabled approach to Cloud RAN test automation, showing the potential of NLP techniques to automatically select test cases from instructions. Additionally, Yang et al. (2022), investigated a low-cost approach for test case generation by leveraging the GPT-3 engine, which concluded that their framework *TestAug* saves the manual efforts in creating the test suites.

Moreover, there are several survey papers on text generation leveraging pre-trained language models (PLMs). For instance, El-Kassas et al. (2020) studied the current application of PLMs to the field of text summarization. Zaib et al. (2021) examined the use of PLMs with a focus on question-answering systems with dialog systems. However, these studies did not address the core approach, which is text generation (Li et al., 2022).

Chapter 4
Data

In this study, we extracted the dataset from the company's documentation. It comprises feature and service documentation and their corresponding test case documents, as an initial dataset was not provided. The documents were available on the internal network as web pages, and to collect the needed data we had to scrape the document pages. We examined three approaches to use as scraping methods, the first option was to manually download HTML screenshot dumps of the web pages containing the documents. The second approach was to scrape the pages through different internal APIs which could retrieve the HTML contents of the pages or lastly use the Python library requests. We initially chose to proceed with the first option to ensure an easy, quick start and to avoid API and authentication complications in the beginning phase of data collecting.

The documents that contained the sought-after data exhibited categorical distinctions, characterized by variations in structure, content, and interrelations among them. A feature document provides a comprehensive overview of a new feature, including its purpose, applications, use cases, and limitations. In contrast, a service document delves into comprehensive details regarding the functionality and configuration of a service, exhibiting greater complexity and having more technical content than a feature document.

The first observation we made was that the quantity of test case documents exceeded that of feature and service documents by a significant margin. This implied that a feature/service document could have multiple corresponding test cases and that some test cases do not have an obvious mapping, see Table 4.5.

4.1 Data Collecting

As aforementioned, we initially collected the data by manually downloading HTML snapshot dumps of the web page containing the documentation. We emphasize that the selected approach was only used initially as, during the working process, we revised the data collecting step in different phases when more data needed to be scraped and the techniques differed

in the different phases.

The second objective after the scraping was to parse the HTML contents to extract the relevant data for our dataset. For this, we utilized the Python library BeautifulSoup (Richardson, 2024) to navigate through the HTML contents and extract the desired data.

4.1.1 Phase I

In the first phase, we only applied the data-collecting process to the feature documents and their corresponding test case documents. The reason we prioritized the feature documents initially was based on their lower complexity and greater ease of parsing. This enabled us to swiftly acquire an adequate dataset with a sufficient size to use for the first iteration of examining the capabilities of the model, which was our main objective at this stage of the study.

Authentication was necessary to access the feature documents, which rendered the request library in Python unusable and did not have an API. Therefore, we manually downloaded HTML snapshot dumps of the documentation pages, which we later parsed using the Python library BeautifulSoup to extract relevant information. In the case of the feature documents, there were only five subheadings that we deemed to have relevant information, the subheadings were: description, purpose, usage instructions, limitations, and decisions.

The database containing all the test case documents had an API which we utilized in combination with cURL (Stenberg, 2024) to filter and extract the relevant test cases related to the acquired feature documents. We identified that the majority of the contents in the test case documents were relevant, therefore all the contents were extracted and parsed (see Figure 4.1).

4.1.2 Phase II

In the next step of the data collection, our focus shifted and targeted the service documents. To access these documents, we were only required to be connected to the internal network, which made it possible for us to scrape these documents using Python's request library (Reitz, 2023) without running into authentication issues. We used the same method as in Phase I to acquire the related test cases for the service documents.

We found that all the subheadings of the service documents contained valuable information, and therefore when parsing the documents we extracted all content from each scraping. Each document consisted of two main subsections, the guide, and the API sections. The guide section exhibited a descriptive tone, while the API section adopted more technical content, comparable to a specification document. Due to the extended length, increased complexity, and inflated amount of test cases corresponding to a single service document, we decided to further divide each primary subsection into multiple sub-subsections. This enabled us to be more accurate in mapping test cases because every test case could be mapped to a more specific and relevant part of the service document. Those mappings could be considered to become a single sample in the dataset.

Further dividing the long service documents into smaller sections helped us evade future problems regarding the model's limited input window size.

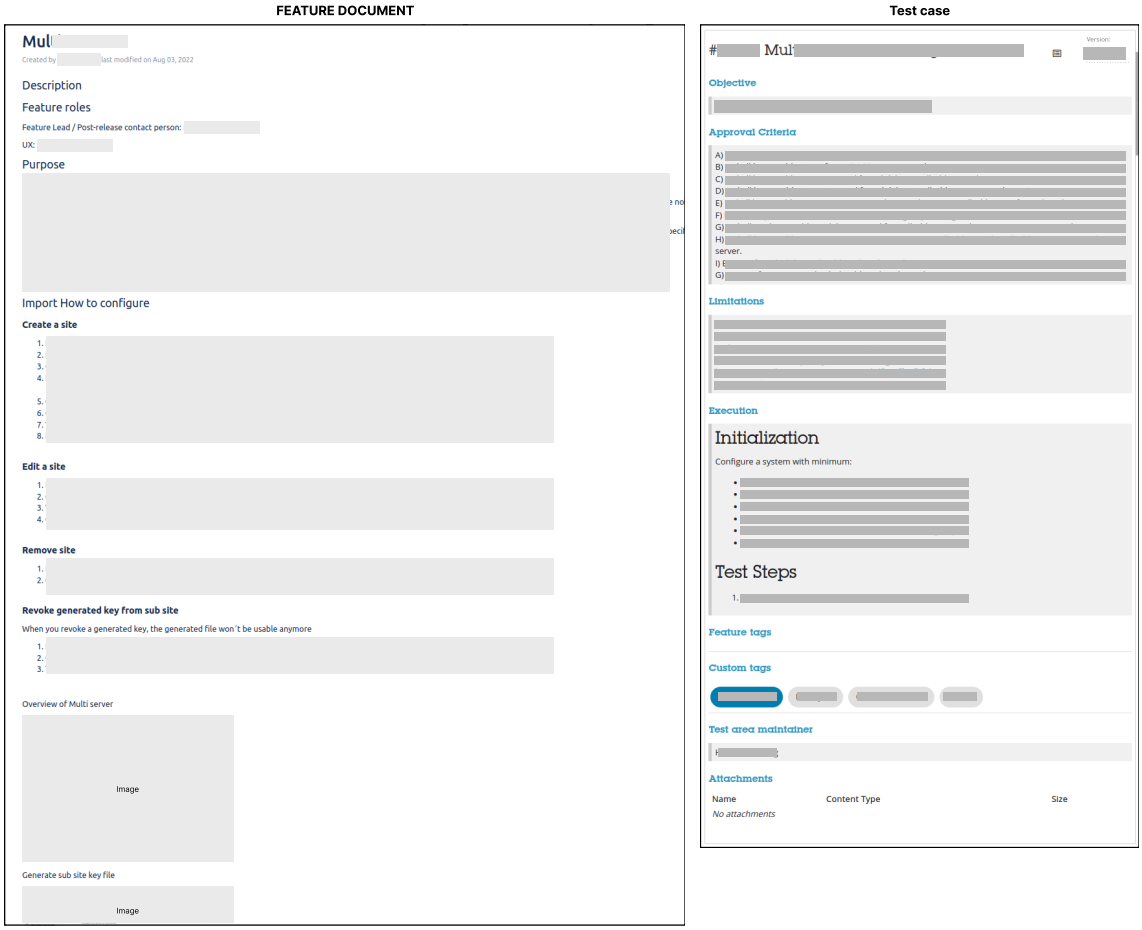

Figure 4.1: Visual example of a feature document with one of its corresponding test case documents.

4.1.3 Phase III

In the previous data-collecting phases, based on a feature and service document, we tried to find relevant test case documents. This resulted in test cases remaining unlabeled. In this phase, based on these unlabeled test case documents, we searched for documents that could be mapped to some of these test cases, not necessarily feature or service documents.

4.2 Exploratory Data Analysis

Exploratory Data Analysis (EDA) consists of several techniques including data cleaning, univariate analysis, bivariate analysis, and multivariate analysis. These techniques can include statistical measures such as mean, median, standard deviation, variance, correlation coefficient, and various graphical methods like histograms, box plots, scatter plots, heatmaps, and pair plot matrices.

According to Otero-Escobar and Velasco-Ramírez (2023), exploratory data analysis is an approach to extract the information contained in the data and summarize the main characteristics of the data. It is considered a crucial step in any data science project and falls in the

second phase of understanding the data.

The insights gained from performing thorough EDA can lead to making informed decisions regarding feature selection, transformation, engineering, or even removal. This ensures better model performance, reduced complexity, and improved interpretability.

4.2.1 Phase I

We conducted EDA on the procured feature documents along with their associated test cases, utilizing various Python libraries, encompassing Seaborn (Waskom, 2021), Sweetviz (de Kock et al., 2022), and Pandas (Wes McKinney, 2010). We introduced a fresh feature description highlighting the primary focus – analyzing data centered around word count per section and subsection – to initiate our investigation. This descriptive approach allowed us to scrutinize and extract valuable insights from the dataset effectively (see Table 4.1).

Table 4.1: Feature descriptions with their corresponding shortcuts

Shortcut	Feature Description
fd_len	The length of the feature document
fd_desc	The length of the 'Description' section in the feature document
fd_purp	The length of the 'Purpose' section in the feature document
fd_howto	The length of the 'How to use' section in the feature document
fd_deci	The length of the 'Decision' section in the feature document
fd_limit	The length of the 'Limitations' section in the feature document
n_testcases	The number of test cases the feature document has
tc_len	The length of the test case
tc_obj	The length of the 'Objective' section in the test case
tc_appcri	The length of the 'Approval Criteria' section in the test case
tc_limit	The length of the 'Limitations' section in the test case
tc_exc	The length of the 'Execution' section in the test case

Subsequently, we generated a heatmap to examine the correlation coefficients of these feature descriptions (Figure 4.2). From the heatmap analysis, we gleaned the following insights concerning the correlations:

Feature documentation relation with their subsections: There is a significant positive correlation between the length of feature documentation (measured by the number of words) and the subsection "How to". This correlation is somewhat lower with the subsection "Purpose" and even less pronounced with the subsection "Limitations". Conversely, the length of feature documentation shows a modest positive correlation with the subsection "Decision" and a less pronounced correlation with "Description".

Test cases in relation with their subsections: There is a significant positive correlation between the length of the test case (measured by the number of words) and both subsections "Objective" and "Approval Criteria". This correlation is somewhat lower with the subsections "Limitations" and "Execution".

Feature documentation relation with test cases: The length of feature documentation correlates positively with all test case properties, but the correlation is small.

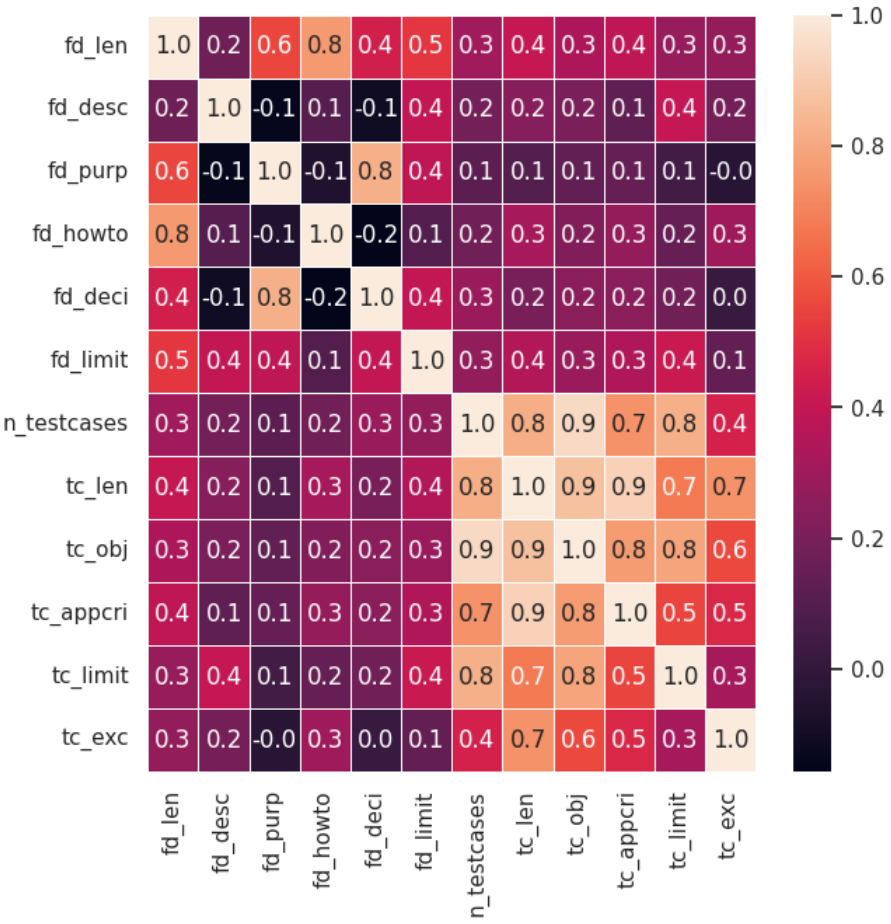

Figure 4.2: Correlation heatmap between the features in phase I.

Additionally, we created a visual representation displaying the linkage between the length of feature documentation and all other features to acquire deeper knowledge and understanding (Figure 4.4). Findings indicated that no discernible effect (correlation value of 0.32) exists between the length of the feature documentation and the number of generated test cases for a specific feature. Furthermore, the "Purpose" subsection within the feature documentation also appears unaffected by this relationship (Figure 4.3). Besides, the distribution patterns (Figure 4.4) for both test case length and feature documentation length reveal diversity. In feature documents, the central tendency typically spans around 408 words, while for test cases, the middle value falls around 150 words.

Moreover, we used bar plots to gain insights into the distribution of feature documentation subsections and the size of each subsection (Figure 4.5). The same approach was applied to test cases (Figure 4.6). The analysis revealed that in existing feature documentation, the predominant content is associated with the "How to" subsection, with a relatively lesser emphasis on the "Purpose" subsection. Meanwhile, in existing test cases, the predominant content aligns with the "Approval Criteria" subsection, with a comparatively lesser emphasis on the "Execution" and "Limitation" subsections. Lastly, we took a deeper look inside all subsections to get a better understanding of them, for this we used box plots (see Table 4.2).

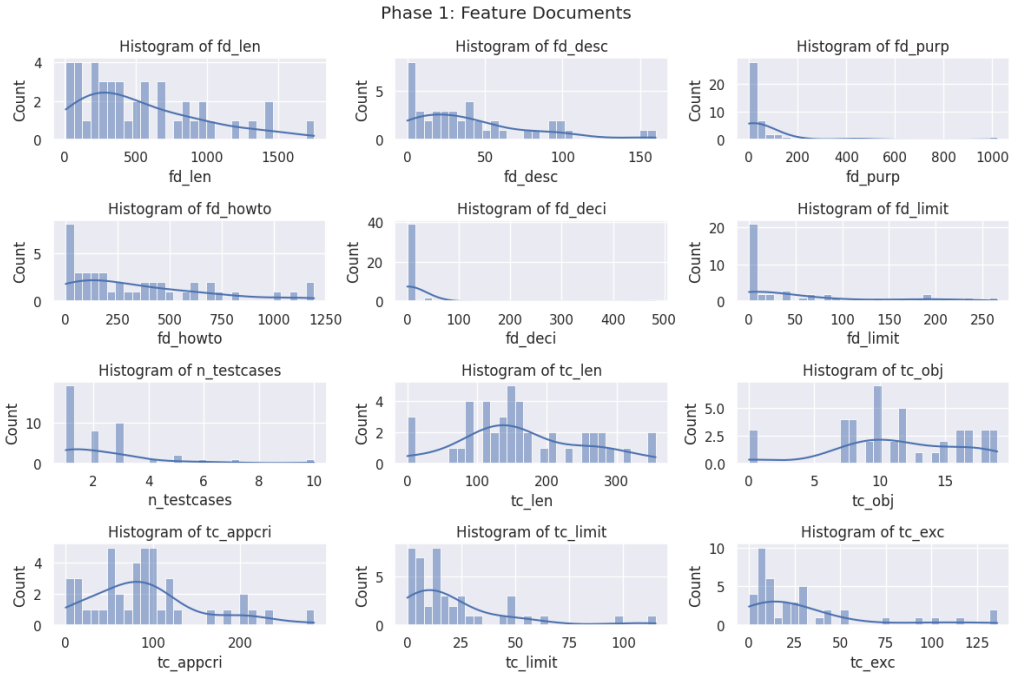

Figure 4.3: Histogram plots of all features.

Table 4.2: Box plot for subsections for both feature documents and test cases.

Boxplot parameters	Box	Median	Minimum	Maximum	Outliers
Feature description	10-60	30	0	100	2
Feature purpose	0-50	≈ 0	0	≈ 150	5
Feature limitations	0-90	≈ 10	0	≈ 205	2
Feature how to use	75-550	≈ 240	0	≈ 1200	0
Feature decision	0	0	0	0	5
Test objective	10-22	≈ 15	0	≈ 35	2
Test approval criteria	90-190	≈ 130	0	≈ 350	3
Test approval limitation	10-40	≈ 15	0	≈ 80	3
Test execution	5-50	≈ 30	0	≈ 110	5

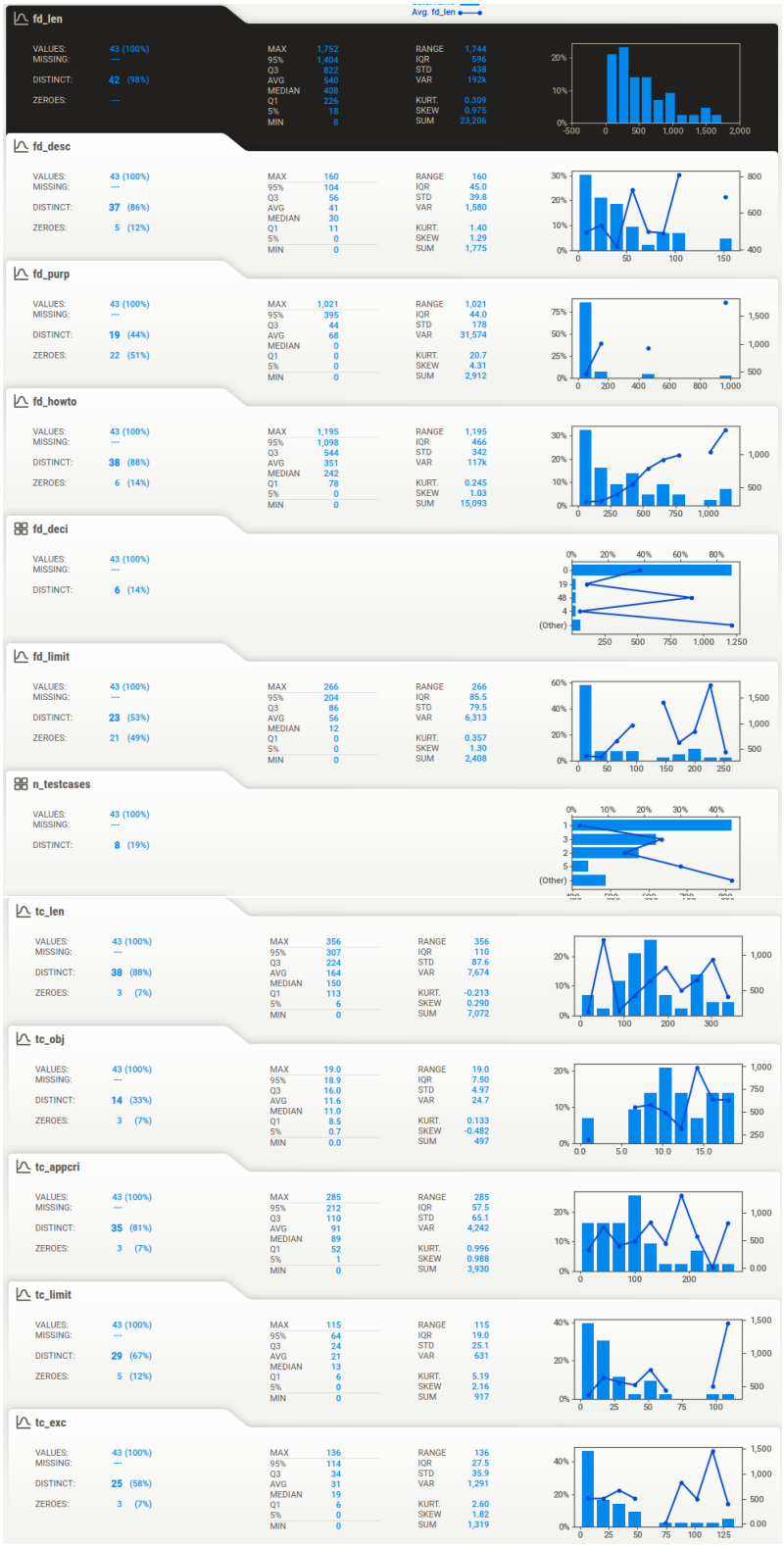

Figure 4.4: Distribution patterns for the length of feature documentation and its relation to all other features using Sweetviz.

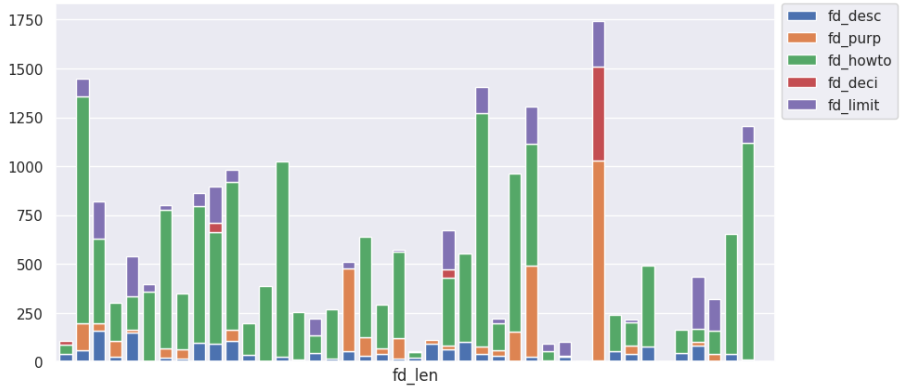

Figure 4.5: Distribution of feature documentation subsections and the size of each subsection.

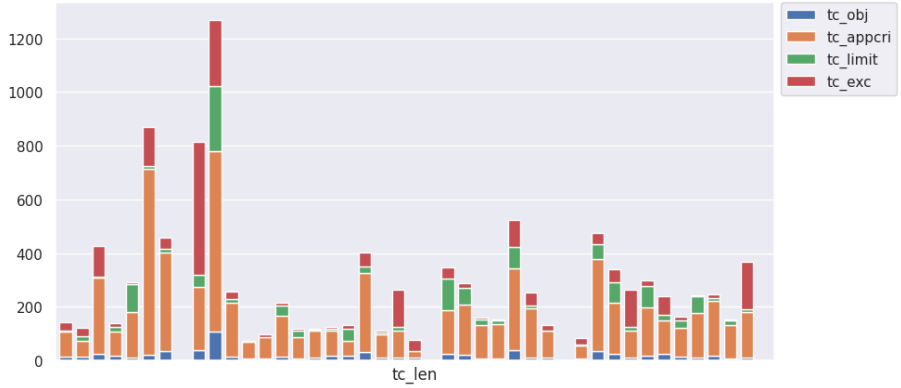

Figure 4.6: Distribution of test cases subsections and the size of each subsection.

Figure 4.7: Phase I: Distribution of feature documentation and test cases with their subsections and the size of each subsection.

4.2.2 Phase II

In this phase, mirroring Phase I, we performed EDA on the acquired service documents and their respective test cases, utilizing the previously mentioned libraries. We applied the same feature description methodology; however, service documents lack subsections found in feature documents. Our focus was exclusively on the segmented sections, centering our analysis once more on word count. Similarly, the corresponding test cases employed identical features, given their uniform structure (see Table 4.3).

Table 4.3: Feature descriptions with their corresponding shortcuts

Shortcut	Feature Description
service_len	The length of the service document
n_testcases	The number of test cases the service section has
tc_len	The length of the test case
tc_obj	The length of the 'Objective' section in the test case
tc_appcri	The length of the 'Approval Criteria' section in the test case
tc_limit	The length of the 'Limitations' section in the test case
tc_exc	The length of the 'Execution' section in the test case

In the following step, we generated a heatmap to examine the associations of both categorical and numerical coefficients of these feature descriptions (Figure 4.8). From the heatmap analysis, we gleaned the following insights:

Service documentation relation with test cases: The length of service documentation correlates positively with all test case properties, but the correlation is small.

Test cases in relation to their subsections: There is a significant positive correlation between the length of the test case and its subsections.

Additionally, we created a visual representation displaying the linkage between the length of service documentation and all other features to acquire deeper knowledge and understanding (Figure 4.9). The results revealed that there is no noticeable impact between the length of the service documentation and the quantity of generated test cases for a particular service (correlation value of 0.21). Besides, the distribution patterns (Figure 4.9) for both test case length and service documentation length reveal diversity. In service documents, the central tendency typically spans around 1431 words, while for test cases, the middle value falls around 1091 words.

Furthermore, we employed bar plots to explore the distribution of test cases and the sizes of their subsections (Figure 4.10). The examination unveiled that within the current test cases, the predominant content corresponds to the "Execution" subsection, with a relatively lower emphasis on the "Approval Criteria" subsection.

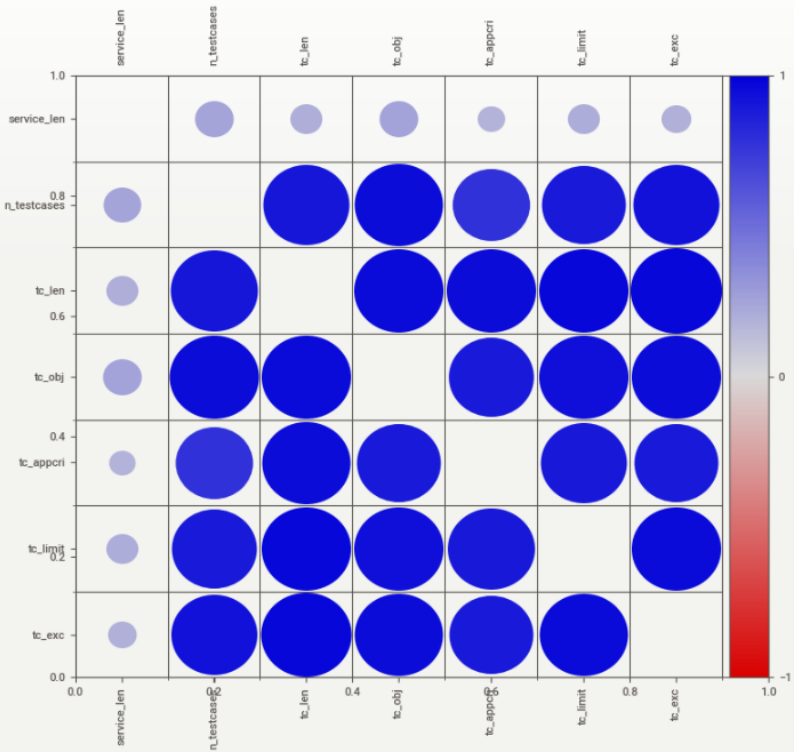

Figure 4.8: Associations heatmap in phase II: Squares are categorical associations (uncertainty coefficient & correlation ratio) from 0 to 1. The uncertainty coefficient is asymmetrical, (i.e., ROW LABEL values indicate how much they provide information to each LABEL at the TOP). Circles are the symmetrical numerical correlations (Pearson's) from -1 to 1. The trivial diagonal is intentionally left blank for clarity (de Kock et al., 2022).

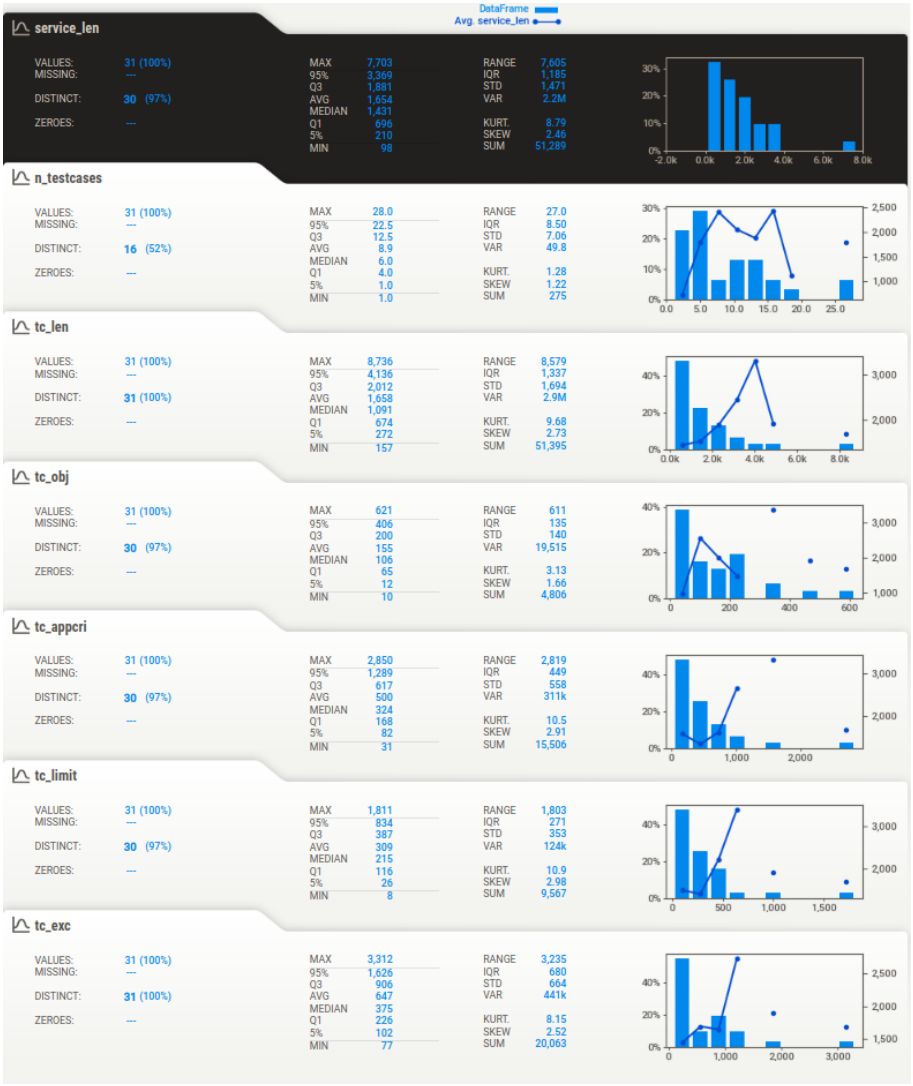

Figure 4.9: Phase II: Distribution patterns for the length of feature documentation and its relation to all other features using Sweetviz.

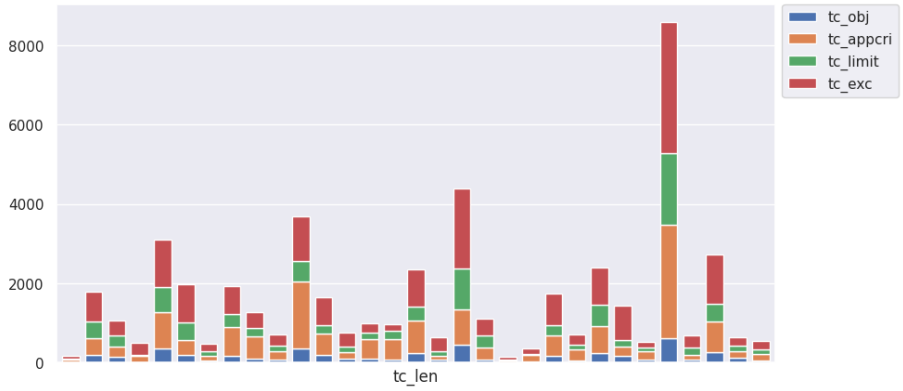

Figure 4.10: Distribution of test cases subsections and the size of each subsection.

4.2.3 Phase III

In this stage, akin to Phase II, we conducted EDA on the recently obtained specification documents and their corresponding unlabeled test cases, employing the previously mentioned libraries. We implemented the same feature description methodology utilized in Phase II (see Table 4.4).

Table 4.4: Feature descriptions with their corresponding shortcuts

Shortcut	Feature Description
specification_len	The length of the specification document
n_testcases	The number of test cases the service section has
tc_len	The length of the test case
tc_obj	The length of the 'Objective' section in the test case
tc_appcri	The length of the 'Approval Criteria' section in the test case
tc_limit	The length of the 'Limitations' section in the test case
tc_exc	The length of the 'Execution' section in the test case

Then, we generated a heatmap to examine the correlation coefficients of these feature descriptions (Figure 4.11). From the heatmap analysis, we gleaned the following insights:

Specification documentation relation with test cases: There is a small correlation between the length of specification documentation and two properties of test cases (number of test cases, and test case subsection "objectives").

Test cases in relation to their subsections: There is a significant positive correlation between the length of the test case and its subsections.

Additionally, we created a visual representation displaying the linkage between the length of service documentation and all other features to acquire deeper knowledge and understanding (Figure 4.12). The results revealed that there is an impact between the length of the specification documentation and the quantity of generated test cases for a particular specification (correlation value of 0.42). Besides, the distribution patterns (Figure 4.12) for both test case length and specification documentation length reveal diversity. In specification documents, the central tendency typically spans around 484 words, while for test cases, the middle value falls around 277 words.

Bar plots were also used to determine the distribution and sizes of test case subsections (Figure 4.13). During the examination, it was revealed that within the current test cases, a relatively lower emphasis is placed on the "Approval Criteria" subsection than on the "Execution" subsection.

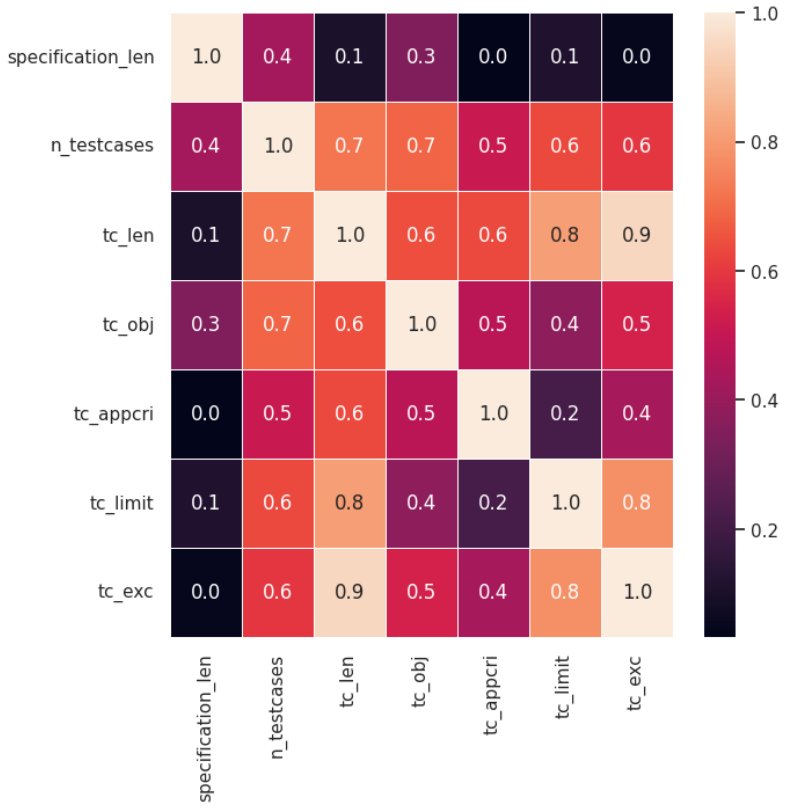

Figure 4.11: Correlation heatmap between the features in phase III.

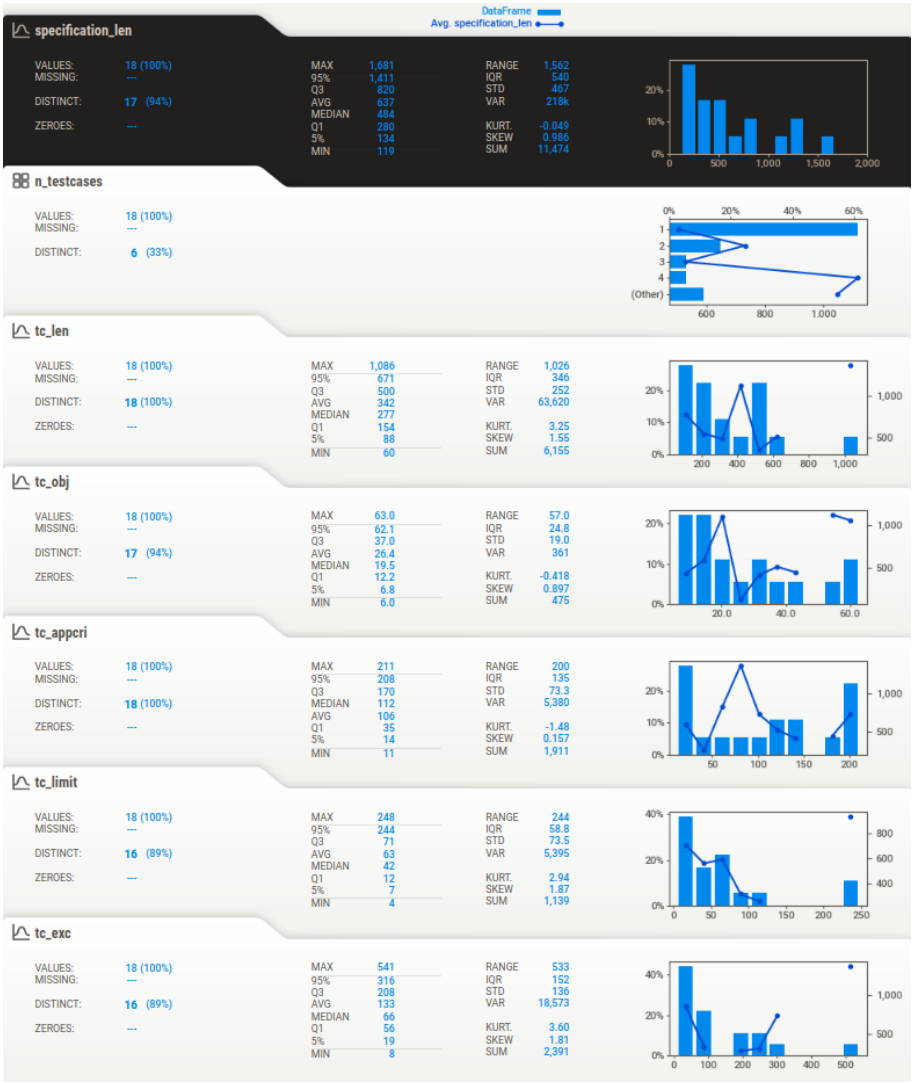

Figure 4.12: Phase III: Distribution patterns for the length of feature documentation and its relation to all other features using Sweetviz.

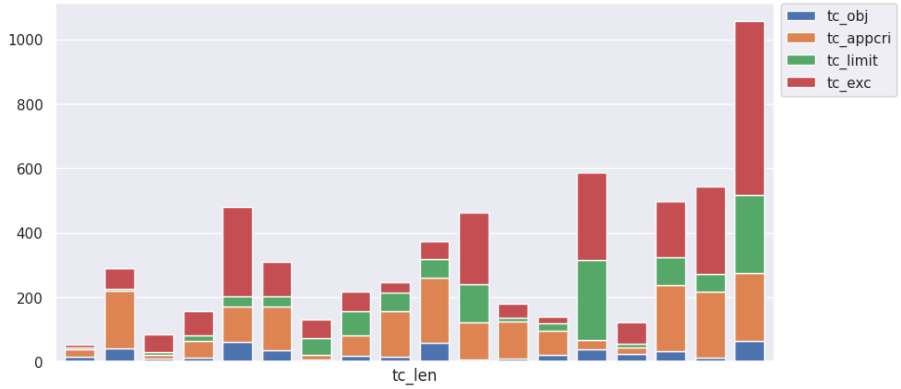

Figure 4.13: Distribution of test cases subsections and the size of each subsection.

4.2.4 Employed Test Cases

We initially had 884 test cases to manage. After performing data filtering and sorting, about half of them remained unlabeled, primarily because clear mappings were not evident (see Table 4.5). We also plotted histograms to analyze distributions for all used test cases with their subsections (Figure 4.14), we noted that the 884 test cases collectively comprise 156,000 words. The majority of these words are found in the "Execution" subsection, totaling 57,486 words, followed by the "Approval Criteria" subsection with 48,353 words, albeit to a lesser extent.

Table 4.5: Summary of employed test cases: Merged test cases refer to situations where multiple test cases are associated with the same document (specification, feature, or service).

All Test Cases			Phase I		Phase II		Phase III	
Total	Mapped	Unlabeled	Mapped	Merged	Mapped	Merged	Mapped	Merged
884	404	480	102	43	275	32	27	18

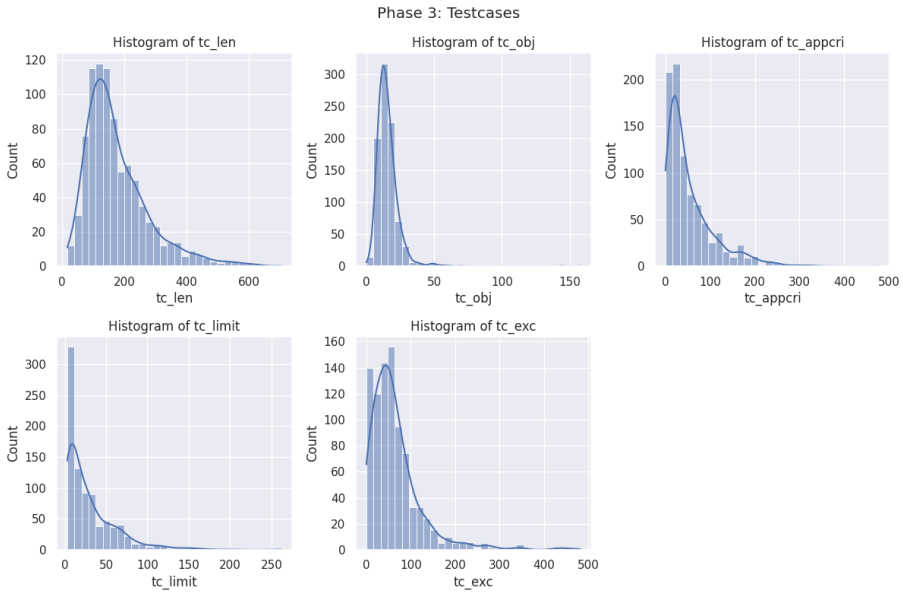

Figure 4.14: Histogram plots of all test cases

4.3 Data Augmentation (DA)

Collecting annotated datasets is a challenging task, time-consuming, and could be impossible in some cases. Thus, strategies for increasing the diversity of training data without explicitly collecting new data have recently seen increased interest, especially in the NLP domain due to the popularity of large-scale neural networks that demand large amounts of training examples.

Despite these challenges, DA strategies have played an essential role in NLP, since they either add slightly modified copies of existing data or create synthetic data. As Feng et al. (2021) discussed, the augmented data distribution should be similar to the original and aim to enhance the performance and reduce overfitting.

DA has different techniques and methods such as *Rule-based* (Choi et al., 2024), *Example interpolation* (Chen et al., 2022), and *Model-based techniques* (Feng et al., 2021). Since we are working with seq2seq models and due to our downstream task, the very last one-mentioned technique is most relevant to us.

Model-based Label-Guided involves using a language model (LLM) and leveraging unpaired or single-sided data samples that contain information about the label, but lack corresponding input data typically found in a supervised learning setting. By using these unlabeled test cases along with appropriate prompts, we can generate corresponding inputs (feature descriptions) through the LLM. Consequently, this allows us to create new data samples by sampling the augmented inputs with their corresponding real labels.

Back-translation has been widely adopted in NLP tasks, and it involves translating a sequence into another language and then back into the original language using a pre-trained language model (Sennrich et al., 2016). Usually, a base pre-trained model is used to be fine-tuned to translate from different specific languages to another. For example, one version of a base model could be trained to translate from English to German and another version of the same base model could be trained to translate from German to English. Two tweaked versions of a base model that translates back and forth between two languages can be utilized to modify a text without changing its context. The difference between the original text which is the input to the back-translation pipeline and the modified version which is the outputted text would be synonym changes and sentence structures, see Figure 4.15 below.

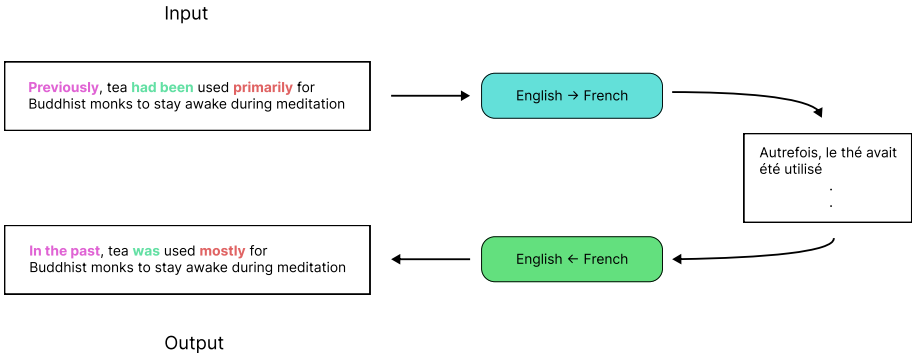

Figure 4.15: Demonstration of a back-translation pipeline

Model-based Document-Level Paraphrasing This approach refers to the process of generating multiple variants of a document using LLM. This technique aims to preserve the overall structure and context of the original document while introducing different versions in its content (Gangal et al., 2022). Since some data samples were too big for the LLM to rephrase in one go, this method was combined with chunking.

4.4 Dataset

Since we aim to generate test cases from documentation, the dataset for fine-tuning the model is required to have a structure where the feature and specification documents are provided as input and their corresponding test cases as output. We investigated different alternatives regarding the format of the dataset where the most interesting options were JSON, CSV, or Pandas (Wes McKinney, 2010) data frame format. Data frames were created using Pandas, while datasets were created using JSON and CSV.

4.4.1 Real Data

Here we collected 4 different datasets (see Table 4.6) according to the previous phases, as follows:

Feature documents to test cases: This dataset contains the data we collected in phase I.

Service documents to test cases: This dataset contains the data we collected in phase II.

Specification documents to test cases: This dataset contains the data we collected in phase III.

Merged documents to test cases: The aforementioned datasets have been merged into one.

Table 4.6: Summary statistics of all real datasets.

Statistics	Phase I		Phase II		Phase III		Merged	
	Doc	TC	Doc	TC	Doc	TC	Doc	TC
Count	43.00	43.00	32.00	32.00	18.00	18.00	93.00	93.00
Mean	3226.69	2663.20	14502.09	11379.34	4291.38	2230.11	7312.47	5578.48
STD	2572.56	2309.03	15999.88	11497.18	3262.05	1671.70	10900.21	8084.63
MIN	66.00	22.00	808.00	1200.00	748.00	422.00	66.00	22.00
25%	1311.00	969.50	5504.75	4734.25	1779.00	1069.25	1827.00	1369.00
50%	2554.00	2245.00	10747.00	7812.50	2998.50	1736.00	3871.00	3030.00
75%	4958.50	3048.50	14988.00	13791.50	6598.00	3269.25	8934.00	6788.00
MAX	10604.00	9682.00	78220.00	61023.00	10560.00	7340.00	78220.00	61023.00

4.4.2 Real & Augmented Data

At this point, after we generated the augmented data from all three aforementioned different DA approaches, we merged all datasets into 3 different datasets as follows:

Small dataset As presented in Table 4.5 we found a total of 884 test cases from which we were able to map 404 to an input document. The remaining 480 test case documents were utilized with the label-guided DA method to generate 480 new data samples. The new data samples were merged with the original dataset, resulting in a dataset with a size of 560 after filtration.

Medium dataset The second dataset was acquired by applying the augmentation method we defined as back-translation above. We applied 8 back-translation pipelines where English was the original language and the following languages were the ones translated to and from Chinese, French, German, Swedish, Italian, Spanish, Russian, and Dutch. This approach resulted in a dataset with a size of 4112 samples when combined with *dataset I* from the previous augmentation approach.

Large dataset In the last method we extracted test case documents from all the departments in the company which amounted to approximately, 10,350 documents. After filtration, we had 9541 test cases from which 480 had already been used in the first DA approach, meaning we had 9061 test case documents at our disposal to apply the Model-based Label-Guided augmentation technique to expand our dataset to approximately 9600 data samples.

It is also worth noting that the test dataset consists of 76 samples taken from the real data is the same every time to be able to maintain comparable results.

Table 4.7: Summary statistics of all datasets

Dataset	Shape	Dataset Split
Small	(560, 2)	train: 387, validation: 97, test: 76
Medium	(4112, 2)	train: 3228, validation: 808, test: 76
Large	(9541, 2)	train: 7569, validation: 1893, test: 76

Chapter 5
Architecture

In this chapter, we will delve into the architecture we used in our study. In the first section, we will discuss text processing and generation. Further, we present the numerical representation methods; Bag of Words, N-grams, and Word Embeddings. In addition, the next section focuses on language modeling and text generation by bringing out the algorithms and techniques it involves. We continue by narrowing down on the Transformer's architecture, explaining its attention mechanism and encoder-decoder framework. The following section analyzes deep into pre-trained language models where different kinds of architectures such as masked, causal, and prefix language models are considered for text analysis. Finally, we introduce Mistral-7B together with its architecture, which demonstrates how it exploits pre-trained language models to improve NLP capabilities.

5.1 Text Processing and Generation

NLP systems rely on text processing and generation to understand, manipulate, and generate human language text. As part of text processing, a wide range of tasks are carried out, such as recognizing words, sentences, and linguistic structures and labeling them with relevant linguistic attributes, to extract meaningful information from raw text data. In this section, we'll briefly introduce a few tasks in modern NLP to make the next section more understandable.

5.1.1 Numerical representation of Text

The majority of NLP tasks rely on converting textual data into numerical representations to assist the transformation of raw text data into a format in which machine learning models can understand and process effectively. Encoding the semantic and syntactic information extracted from text documents into numerical vectors enables computational algorithms to operate on them. The NLP field began with the exact matching technique, where Context

Free Grammar (CFG) [1] was used for analysis. In the early days of search engine development, complex nested if-then rules were used to build search engines. Further advancements led to approximate matching, which ignored errors up to a certain point. As research progressed, statistical approaches were used to focus on word frequency. Compared to grammar-based techniques, these methods were easy to implement and improved model accuracy (Patil et al., 2023). We will explore a few methods:

Bag of Words (BOW) In this method, the corpus is displayed as a matrix where each sentence is a row and each unique word is a column (Harris et al., 2017). The number of rows is determined by the number of sentences in the corpus, and the vocabulary size of the corpus determines the number of columns. This matrix has dimensions according to the number of sentences in the corpus. By using rows to represent each sentence or document in a corpus, similarity scores between them can be calculated.

Nonetheless, as discussed by Patil et al. (2023), this representation is contingent on precise matching, indicating that document similarity only works when exact words are employed. This way, it may produce an inaccurate similarity score for documents containing the same information but using different words or synonyms.

N-Grams This method shifts the focus from individual words to multi-word tokens, and captures the ordering present in the window of context. Character 'N' indicates the size of that window. As Katz (1987) proposed, N-gram was the first technique to define a window to capture the ordering among words, therefore, it becomes effective when certain groups of words carry more weight or different meanings when they appear together. Furthermore, N-grams that occur rarely or too frequently do not carry any important meaning, thus, filtering is needed to eliminate them.

Word Embeddings This method comes to hand due to the importance of the semantic relation between words and the widespread use of representing words as low-dimensional dense vectors. The embeddings are converted into a continuous vector space. Each word is represented by a set of real numbers (see Figure 5.1), aiming to understand the meaning of words based on their context.

In modern NLP, word embeddings can predict surrounding words and capture their relationships. Surrounding words can affect the given word and the meaning of it, which could lead to them being context-sensitive (Patil et al., 2023). In addition, word vectors capture their meanings and represent them using dense floating-point values, which represent both the semantics and syntactic aspects of the word. These vectors as Patil et al. (2023) claimed have a length between 100 and 500 dimensions.

Text embeddings are constructed in many ways, such as the early approaches *word2vec* (Mikolov et al., 2013) and *GloVe* (Pennington et al., 2014), which build vector representations of documents by using document statistics (Romanus Myrberg and Danielsson, 2023). Nowadays, embeddings are often generated by large language model encoders based on the transformer architecture, and then the learned embeddings are used as pre-trained embeddings, which reduces the computational cost (Vaswani et al., 2017).

[1] Context Free Grammar (CFG) is a formal grammar with production rules that can be applied to a nonterminal symbol without considering its context. (Cremers and Ginsburg, 1975).

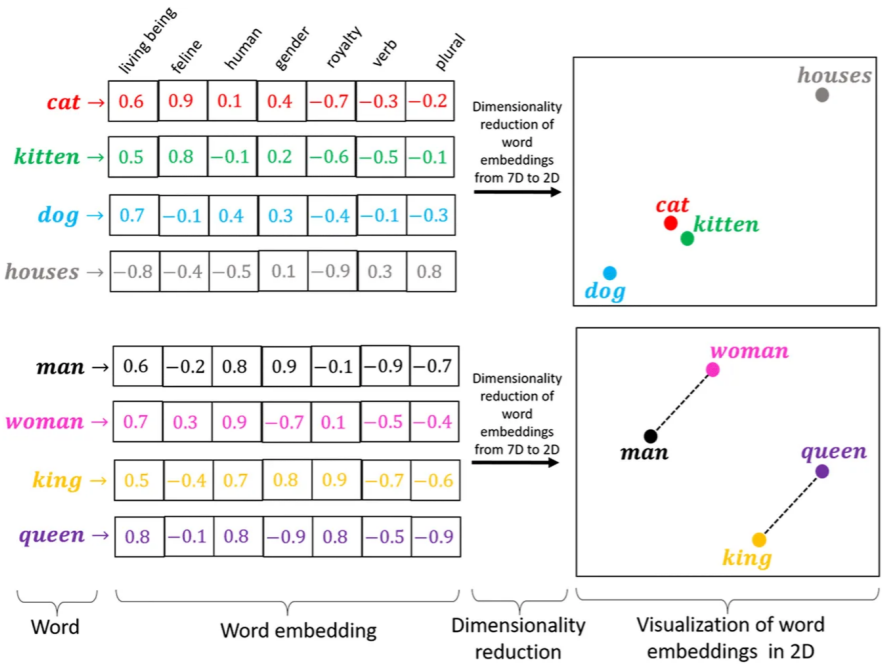

Figure 5.1: Illustration of word embeddings and their visualization in 2D (Alvarez, 2017)

5.1.2 Text Generation and Language Modeling (LM)

Text generation, the process of producing easy-to-comprehend and contextually relevant text from a given input, is a critical task within NLP. It heavily depends on language models, with transformer-based architectures being especially prominent in this domain.

Language Modeling (LM), as an unsupervised task in NLP, serves as the cornerstone for many text generation tasks, facilitating the generation of human-like text with rich contextual understanding. However, a fundamental problem for language modeling is working with high-dimensional data *curse of dimensionality* [2] (Leygonie et al., 2023) which seems to be obvious when modeling the joint probability distribution between multiple discrete random variables (such as words in a sentence) Bengio et al. (2000).

Generally, a text can be modeled as a sequence of tokens $\mathbf{t} = \langle t_1, \ldots, t_j, \ldots, t_n \rangle$ (Li et al., 2022), where each token is drawn from a vocabulary V (set of symbols). Therefore, it is possible to represent a statistical model of language by the conditional probability of the next word given all the previous words, since

$$\hat{P}(w_T^1) = \prod_{t=1}^{T} \hat{P}(w_t | w_{t-1}^1) \tag{5.1}$$

5.1: Conditional probability of the next word given all the previous words, where w_t is the t-th word, and $w_j^i = (w_i, w_{i+1}, \ldots, w_{j-1}, w_j)$

[2] curse of dimensionality: when dealing with high-dimensional data, many supervised pattern recognition methods may not be able to capture all relevant information. As the number of dimensions increases, exponentially more data is required to maintain a similar density.

In addition, we can iteratively sample tokens $\hat{t}_i$ from the learned distribution $p_\theta(t_i|t_{<i})$. , $t_{<i}$ denotes all the tokens sampled before $\hat{t}_i$. In each iteration, we sample a token based on the conditional probability distribution given by the model and insert the sampled token into the sequence. By repeatedly selecting tokens based on learned probabilities, we can gradually generate text one token at a time (Radford et al., 2019).

5.2 Transformer's Architecture

A Transformer is a form of deep neural network first introduced in Vaswani et al. (2017). Its architecture revolutionized the world of NLP as transformer-based models surpassed previous state-of-the-art approaches by such a huge margin in performance, which led to all the recent cutting-edge models being based on transformer architecture (Gillioz et al., 2020). The initial application of the transformer architecture was on machine translation, and today we can see transformers being applied in most, if not all, NLP tasks. The factor that distinguished transformers from the rest was the attention mechanism of the model architecture. Vaswani et al. (2017) describe that to the best of their knowledge transformers were the first model architecture that solely relied on the attention mechanism to establish global dependencies between input and output which resulted in a significant increase in efficiency, parallelization, and cutting-edge machine translation.

5.2.1 Attention Mechanism

Niu et al. (2021) analogously describe the attention mechanism in the neural networks domain as the focused attention we humans have when we consciously and actively focus on a particular assignment. The application of the attention mechanism has become more prevalent in neural network architectures (Niu et al., 2021).

The transformer architecture uses self-attention. Vaswani et al. (2017) describe self-attention as an attention mechanism that uses the relation between different parts of an input sequence to generate a representation of the sequence. From the input sequence, a query (Q) and key-value (K, V) pair are derived and mapped to an output sequence. All these variables are vectors where the output is the weighted sum of V and the weight assigned to a single V is determined by the compatibility function of the Q with the corresponding K value.

Scaled Dot-Product Attention (SDPA). Vaswani et al. (2017) introduced "Scaled Dot-Product Attention" as the attention function for the transformer architecture. Their approach allowed for higher efficiency and effectiveness in attention computation by scaling the dot product of Q and K to further apply the softmax function to obtain the attention weights (see Equation 5.2). This enabled parallelization and more stable gradients during training which meant increased performance compared to the previous most commonly used attention functions, namely additive and multiplicative dot-product attention in diverse NLP tasks. Vaswani et al. (2017) subsequently clarifies that the algorithms of multiplicative dot-product and SDPA are identical except for the scaling factor. In the case of additive attention it has a similar complexity as SDPA in theory but due to highly optimized matrix multiplication is SDPA significantly faster and memory-efficient.

$$Attention(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{5.2}$$

5.2: Scaled Dot-Product Attention function (Vaswani et al., 2017)

Multi-Head Attention. Vaswani et al. (2017) demonstrated that instead of applying single-head attention that uses the same dimension for Q, K, and V, employing multi-head attention enables the model to target different aspects of information across different parts of the data. This improves the model's ability to recognize the semantic relation between the words in the input. Multi-head attention is achieved by linearly projecting Q, K, and V multiple times repeatedly using various learned linear projections to d_k, d_k, and d_v dimensions. Further, the attention function is simultaneously applied to each of the various projected Q, K, and V, which results in an output with the dimension d_v (see Figure 5.2).

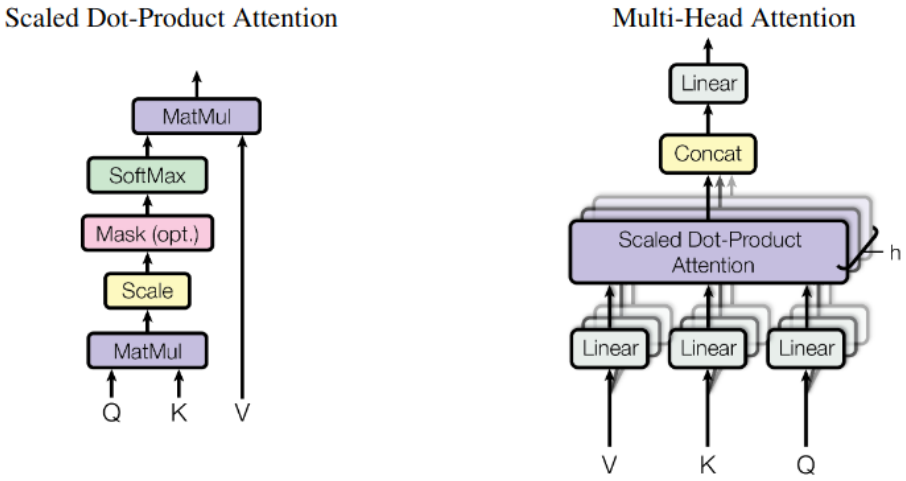

Figure 5.2: Visualization of the Multi-head Attention block (Vaswani et al., 2017)

5.2.2 Encoder-Decoder

The architecture that transformers are based on differs from the previous architectures used for sequence-to-sequence models. Formerly, the most common approach was to use recurrent neural networks or convolutional neural networks for those kinds of tasks. When Vaswani et al. (2017) introduced the transformer architecture which comprises two main parts: an encoder and a decoder, it revolutionized the playing field of model architectures as it enabled more effective and efficient learning and combined with the self-attention mechanism resulting in the capability to capture global dependencies between the input and output sequence.

Encoder. The main responsibility of the encoder component is to create contextualized embeddings based on the input sequence, $x = (x_1 ... x_n)$ which consists of symbol representations in a sequence. By utilizing the self-attention mechanism, this module carefully examines the tokens in the input sequence. Subsequently, it establishes connections between tokens from parts of the sequence to emphasize their relevance within the context. Through this

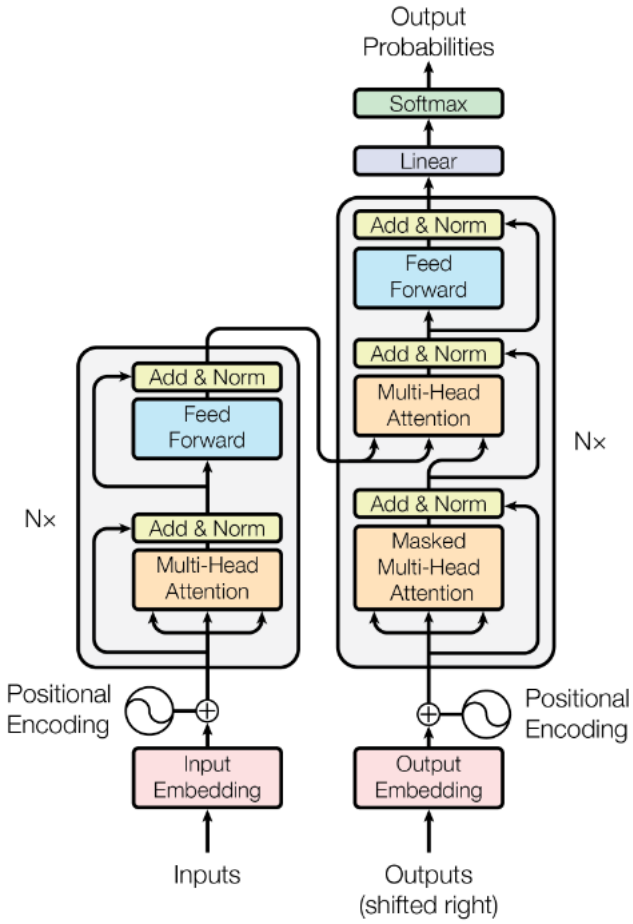

Figure 5.3: Visualization of the encoder-decoder architecture. The module to the left is the encoder and the one on the right is the decoder (Vaswani et al., 2017)

process, the encoder module generates embeddings that capture details, about the input sequence. The resulting continuous representation sequence $z = (z_1 ... z_n)$ is then passed on to the decoder.

Vaswani et al. (2017) describe that the encoder is composed of six stacked layers that are identical and within every layer, there are two sub-layers, namely a multi-head self-attention mechanism block and a fully connected feed-forward network. Each of the sub-layers is then followed by a normalization layer (see Figure 5.3).

Decoder. The decoder's main objective is to utilize the contextual embeddings generated from the encoder to predict an output sequence that is coherent and contextually appropriate to the input sequence. The module is implemented to behave autoregressive which means the model generates one token, y_i, at a time in the output sequence. This enables it to predict the subsequent token with relevance to its preceding ones. The decoder continuously re-feeds each outputted predicted token, y_i, to maintain the relevant context in which the next generated token y_{i+1} is dependent on to be able to predict a contextually relevant word relative to the input sequence x. The self-attention mechanism used in the encoder module is

also implemented in the decoder, which plays a pivotal role in creating global dependencies between the input and output sequence.

Figure 5.3 shows that the decoder similar to the encoder consists of six stacked layers but differs in the amount of sub-layers it comprises. The decoder has an additional sub-layer besides the fully connected feed-forward network and the multi-attention layer that the encoder also contains. The extra sub-layer is a masked multi-head attention block followed by a normalization block, which is the layer that performs the attention mechanism on the contextual output from the encoder to the decoder. Transformers can guarantee a generated prediction y_i to have solely relied on the known predictions preceding $i : th$ position due to the masked attention and the fact that the incoming contextual embeddings from the encoder are shifted by one position (Vaswani et al., 2017).

5.3 Large Language Models (LLMs)

In this section, we discuss the pretraining of LLMs, which is an extension of our previous discussions on their architecture and functionality. Pretraining is an essential part of LLMs, which is the basis of their capability to comprehend and produce natural language. LLMs go through the process of unsupervised learning, during which they learn to detect linguistic patterns and semantic relationships that are part of the data that they are processing. This intermediate stage of pretraining is very important, as it provides LLMs with the knowledge and context that they need to do a lot of different tasks well. In the next subdivisions, we discuss the pretraining objectives, dataset selection, and fine-tuning procedures, thus, giving a brief of the complex processes in the formation of the abilities of the latest language models.

5.3.1 Pre-trained Language Models (PLMs)

Pretrained language models (PLMs) have become increasingly popular in natural language processing in recent years. They are trained on large unsupervised corpora first and then fine-tuned in downstream supervised tasks later on. In response to Transformer and higher computational power, PLM architecture evolved from shallow to deep architectures (Li et al., 2022), like *BERT* (Devlin et al., 2019) and *OpenAI GPT* (Radford et al., 2019).

Extensive research has demonstrated, as Li et al. (2022) discussed, that pre-trained language models can store a vast amount of linguistic knowledge in their parameters and acquire universal and contextual representations of language by utilizing specifically designed objectives, such as masked token prediction [3]. PLMs are therefore generally beneficial for downstream tasks and can prevent training a new model from scratch. Following the success of PLMs in other NLP tasks, researchers have proposed (Raffel et al., 2023) to apply PLMs to text generation tasks with several steps (see Figure 5.4).

[3]Masked language modeling (MLM) first masks out some tokens from the input sentences and then trains the model to predict the masked tokens by the rest of the token (Qiu et al., 2020).

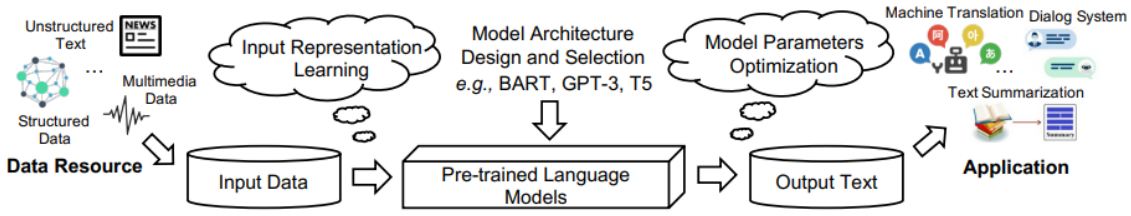

Figure 5.4: A three-step process for using PLMs for text generation is described: input representation learning, model architecture design and selection, and model parameter optimization. (Li et al., 2022)

5.3.2 PLMs Architecture

Transformer-based encoder-decoders or single Transformers are the foundation of existing PLMs for text generation. PLMs, such as *GPT-3* (Brown et al., 2020), use a single Transformer, the decoder only, to perform input encoding and output decoding simultaneously (Li et al., 2022). This encompasses three key variations: masked LMs, causal LMs, and prefix LMs, varying in the strategy of attention masking.

Masked Language Models Masked LMs use a full-attention Transformer encoder. Usually, full-attention models are trained to predict masked tokens using bidirectional information for masked language modeling (MLM). BERT (Devlin et al., 2019) represents the most representative model (see Figure 5.5).

However, masked LMs are seldom used for text generation tasks due to the discrepancy between pre-training and downstream generation tasks. Yang et al. (2020) conclude that BERT's reliance on manipulating input with masks neglects dependency between masked positions and results in a pretrain-fine-tune discrepancy. Masked LMs are more frequently used as encoders for text generation, leveraging their excellent bidirectional encoding abilities, as Rothe et al. (2020) propose.

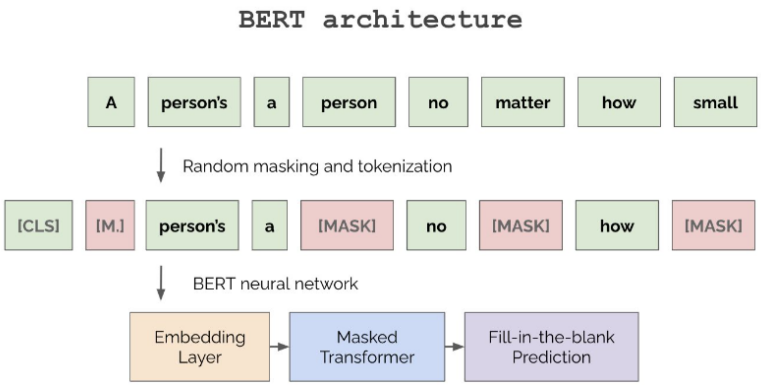

Figure 5.5: Illustration of BERT architecture as masked transformer (Laskin et al., 2022)

Causal Language Models Likewise to the Transformer decoder, causal LMs rely on the diagonal mask matrix and are designed for language modeling, predicting the likelihood of

certain words in a sentence occurring. Therefore, causal LMs are straightforward for text generation, predicting the next word based on all previous words (Li et al., 2022).

In previous studies, *GPT* (Radford et al., 2018) was the first causal LM for the text generation task (see Figure 5.6). Following that, *GPT-2* (Radford et al., 2019) investigated the transfer capability of language models for zero-shot generation tasks, emphasizing the importance of having sufficient data. Moreover, *GPT-3* (Brown et al., 2020) demonstrated that massive model parameters can dramatically improve the downstream generation tasks, with a few examples or prompts.

As mentioned before, causal LMs are simple for text generation, but they have several structural and algorithmic limitations. Since they encode the tokens just from left to right, they ignore the bidirectional information on the input side. Additionally, causal LMs are not intentionally designed for the sequence-to-sequence generation tasks, thus in practice, they do not exhibit high performance in tasks such as summarization and translation (Radford et al., 2019).

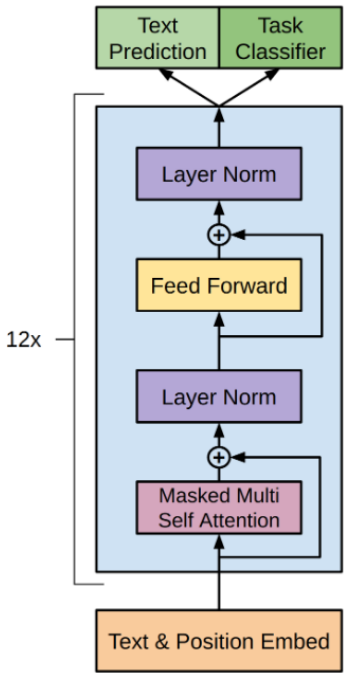

Figure 5.6: The GPT-1 architecture laid out by Radford et al. (2019). It is composed of 12 stacked decoders. Each decoder contains self-attention, followed by a position-wise feed-forward network, with normalization in between (Al-Hossami and Shaikh, 2022).

Prefix Language Models Prefix LMs, functioning on a single Transformer, use a bidirectional encoding scheme for input and a natural left-to-right generation pattern for output. Through the mixture attention mask, the input text tokens can relate to each other, while the target text tokens can only attend to all input tokens and previously generated tokens (Li et al., 2022).

UniLM (Dong et al., 2019) was the first prefix LM (see Figure 5.7). It employed a prefix attention mask to deal with conditional generation tasks, compared to causal LMs, similar

to the encoder-decoder architecture. UniLMv2 (Bao et al., 2020) and GLM (Du et al., 2022) enhanced vanilla prefix masking strategy by introducing permuted language modeling [4] in XLNet (Yang et al., 2020).

After comparing single transformer prefix LMs to transformer-based encoder-decoder LMs, Raffel et al. (2023) concluded that adding explicit encoder-decoder attention is more robust in capturing conditional dependencies, despite the advantages of prefix LMs.

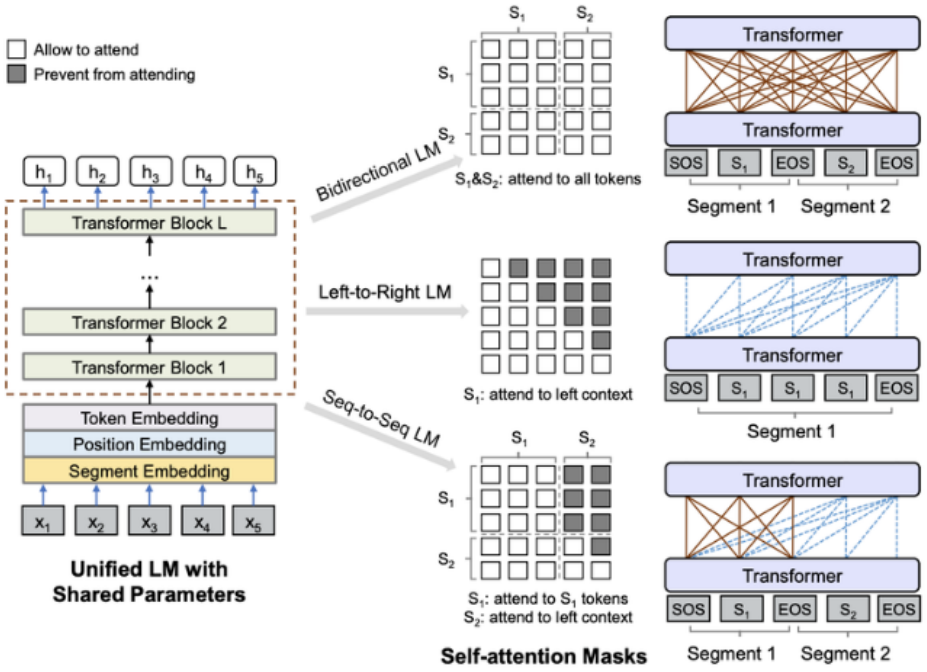

Figure 5.7: Unified LM architecture as illustrated in Catterall et al. (2005).

5.3.3 Fine-tuning Large Language Models

In machine learning, fine-tuning is a technique of adapting a pre-trained model to a new task or use case. It could be considered a variant of transfer learning.[5]

The key idea of fine-tuning is that we save lots of time and resources since it is cheaper and easier to build upon the capabilities of a pre-trained base model which knows relevant to the downstream task than training a new model from a blank slate. Additionally, this is extremely valuable in cases where we are dealing with deep learning models with millions or even billions of parameters, such as LLMs.

Parameter Efficient Fine Tuning (PEFT) is an innovative solution to fine-tune large pre-trained models without requiring extensive computational resources that are needed to

[4] Given a sentence, instead of predicting the next word in sequence (as in traditional LMs), predict a randomly permuted version of that sentence.

[5] Friederich (2017) defined transfer learning as the practice of using an existing model's learned knowledge as the basis for learning new tasks.

fine-tune a whole LLM (Xu et al., 2023). Freezing most layers of the large model and only leaving the parameters in the last few layers for fine-tuning increases efficiency by reducing memory usage and the number of fine-tuning parameters while still maintaining results good enough to be comparable to fully fine-tuned model (Takyar, 2023). PEFT's effective solution enables us to fine-tune a pre-trained model to a smaller dataset to achieve better performances on domain-specific tasks, in our case, test case generation.

Quantized Low-Rank Adaptation (QLoRA) Fine-tuning LLMs today requires a copious amount of GPU memory due to the scale of the models. A rough estimate of memory requirements for loading a model into a GPU with an optimizer is visualized in Table 5.1 below.

Table 5.1: The values are based on fp32 precision training and normal AdamW optimizer (Mittal and Vetter, 2014)

Type	Memory Requirements
Model Weights	4 Bytes × Number of Parameters
Optimizer	8 Bytes × Number of Parameters
Gradients	4 Bytes × Number of Parameters
Forward Activations	High variation, depend on many variables like sequence length

Based on the table above, regular fine-tuning of an LLM with 65 billion parameters needs beyond 780GB of GPU memory, a capacity far more than any single GPU has today (Dettmers et al., 2023). Smaller models with 7 billion parameters require more than 112GB of GPU memory, which is still much more than the capacity of a single standard consumer GPU with between 8 and 16GB of memory. To address the memory issue, Hu et al. (2021) present a solution named Low-Rank Adaptation, LoRA, where the technique freezes the pre-trained model weights and instead infuses trainable parameters into every layer in the architecture in the form of rank decomposed matrices. This method significantly reduces the number of trainable parameters with a factor of up to 10,000 times and can reduce the GPU memory requirement up to 3 times. LoRA is elegantly implemented to have a linear design resulting in no inference latency when the injected trainable parameters are merged with the frozen pre-trained parameters.

Applying LoRA and hypothetically reducing the memory capacity with the highest factor of 3 on a smaller model with 7 billion parameters would still need approximately 10GB of GPU memory to load the model, leaving little to no memory left for fine-tuning on a single standard consumer GPU.

Dettmers et al. (2023) introduced QLoRA which applies multiple techniques designed to lower the GPU memory requirements for the models while maintaining the performance. QLoRA quantizes the weight precisions of the frozen weights from the pre-train model from 32-bit into 4-bit. Through this approach, the weights of a model like Mistral-7B would need approximately 6GB of GPU memory to load into a GPU instead of 28GB. The injected parameters would still have the 32-bit precision to make it possible to fine-tune the model on the infused weights. This approach enables us to fine-tune the model with significantly lower GPU requirements while still maintaining the model's pre-trained knowledge.

Supervised fine-tuning training (SFT training) Wolfe (2023) describes supervised

fine-tuning (SFT) as the training approach employed by LLM alignment, in which a dataset of high-quality model outputs is constructed, and the model is fine-tuned on that dataset directly. The "supervised" part of SFT comes from the fact that the dataset of high-quality model outputs is a set of examples of good model behavior, and the model's fine-tuning process is such that the model learns to write in the same style as these examples.

At a high level, SFT looks similar to language model pretraining, in that both methods have next token prediction as their base training task. However, the major difference is where the data comes from. Instead of pretraining on a vast raw corpus of text, SFT learns from a few dozen examples in each fine-tuning step. On each iteration, a number of examples are sampled, and then the model is fine-tuned to these examples directly.

SFT avoids the risk of models becoming more specialized and losing generic problem-solving ability in comparison to the generic fine-tuning that follows a curriculum. This is because it fine-tunes models to copy a fine-tuned model's style or skill, but not to solve a task.

Hence, SFT naturally becomes an essential method for fine-tuning language models. The fine-tuning process is computationally cheaper with a factor of no less than one hundred than the regular process of pre-training (Wolfe, 2023).

Instruction Fine-tuning Since LLMs are trained on vast text data usually sourced from the internet (articles, scientific papers, blogs, etc) and physical documents (books, newspapers, notes, etc) the fundamental training of the model aims to predict missing or subsequent words in a provided sequence. Processing this multitude of sentences results in the model acquiring a deep knowledge of grammar, context, and semantics, hence being able to predict the missing or subsequent word with high accuracy. However, the initial training of an LLM only makes the model exceptional when its input consists of sequences similar to the sequences in the data it was trained on. The model would generate reasonable outputs if provided sentences from a news article but would perform poorly if the input sequence resembles instructions to guide the model to perform a specific task. This is because the model is trained on informative data where directive-like sentences are not frequent (Zhang et al., 2024).

Instruction fine-tuning is an approach that addresses this deficiency. For a model to accurately perform a specific task by following provided instructions, it needs to be trained on data containing samples that bear similarities to the characteristics of the opted instructions and the nature of the aimed task.

If such a dataset is available, then it could be utilized to adapt the parameters of the model to align with the specific task. This is accomplished by training the model on the instructional dataset on top of its initial training to leverage its prior knowledge from the pre-training. The fine-tuning updates the model parameters through back-propagation during training, where the parameters are adjusted to minimize the loss from the differences between what the model is currently generating and what it is supposed to generate (Khawaja, 2023).

The process of model parameter adjustment based on the instructional dataset is referred to as instruction fine-tuning. This technique has several benefits, firstly leveraging the prior natural language knowledge saves a substantial amount of computational resources and time by avoiding retraining the model from scratch. Secondly, the fine-tuning taps into the model's full potential by making it attuned to the intricacies and nuances of the domain of the particular task. Further, with instruction fine-tuning, it becomes easier to control the model's

behavior and generation. Lastly, instruction fine-tuning requires less available data than regular fine-tuning and training a model from scratch, which approaches a more effective way of specializing a model to a specific domain. Examples of domain-specific tasks are text summarization, text sentiment analysis, and questioning and answering where a model performs either of the tasks based on a set of instructions or requirements (Banjara, 2024).

5.3.4 Prompt Engineering

Prompt engineering is a systematic approach aiming to successively improve the input fed into an LLM, attempting to improve the quality and accuracy of the model output. Simply explained, the core of the approach is to slowly, usually by trial and error, design a set of instructions or a demonstration that is added to the input data that is passed to the model. The prompt is usually added as a prefix to the data samples in the dataset, so the model first processes the instructions in the added prompt and then proceeds to handle the data contents of the sample and tries to generate an output according to the instructions prompt (Arvidsson and Axell, 2023).

Chain Prompting has a straightforward yet powerful idea: using the output of a language model to drive subsequent inputs. The whole process is just a chain of repeated episodes that supply the model with some feedback instructions based on the generated output from the previous prompt. The introduction of this feedback mechanism into the input pipeline of the model in chain prompting is what creates a mechanism for iterative refinement of the model's output for coming closer and closer to the desired outcome. At every step in the chain, the model receives feedback in the form of corrections, guidance, or reinforcement signals, and these are intended to steer the model in the direction of generating outputs that meet criteria or objectives better and more closely. With this continuous refinement process, chain prompting empowers language models to produce outputs that are more contextual coherent, relevant, and accurate (Ramlochan, 2023).

It promises a lot as a method of optimizing pre-trained LLMs as it offers many benefits. Firstly, by iteratively incorporating feedback, chain prompting enables the refinement of generated outputs across iterations. Here, the model gains a chance to learn from its own mistakes and adapts its behavior to better deliver the desired outputs. Further, by implementing a feedback loop, chain prompting allows the model to gain a deeper comprehension of the context and requirements of the task. This improved contextuality allows the model to generate higher-quality outputs. For the same reasons, chain prompting provides a flexible framework, which could be adapted to suit different purposes and objectives. Given the feedback to the model, which may prioritize specific criteria or improve performance in certain use cases, this approach is appropriate for a wide number of tasks (Anthropic, 2024).

Lastly, rather than fine-tuning a model with limited or low-quality data which is a more computationally expensive and complex process the incorporation of feedback iteratively, and chain prompting will enable efficient model optimization.

Single and Multi-shot Prompting A single-shot prompting language model receives only one prompt, which is an instruction followed by one example. The model generates its response from this input only, and its result ideally follows the instruction and the given example. This technique is very specific and efficient, which makes it useful in quick inquiries

or single instances examples. Thus, it may cause a limitation of the model to capture non-repetitive contexts or produce diverse outputs because it acts only according to the provided prompt (Maclaren, 2024).

While with multi-shot prompting, you feed the language model several examples or pieces of details in one prompt. Users engage with the model by presenting various examples or contexts simultaneously. This method allows for the inclusion of diverse and detailed information into a single conversation, which aims to translate into richer and contextual outputs (Maclaren, 2024).

Increasing the context in the input often boosts the coherence of the generated sentences with regard to the desired output, as in multi-shot prompting, where you can add multiple samples to one prompt, causing the model to have a better comprehension and create a closer response. In a similar way, a single-shot prompt that is carefully created and equipped with sufficient context can often enhance the model's answer by directing its concentration to certain aspects of the task (Sivarajkumar et al., 2024).

In the prompt engineering space, there is a need to be aware of the specifics as well as the pros and cons of single-shot and multi-shot approaches for exerting full control of language models in order to attain the intended goals. Adding more context and examples assists the model in creating outputs that are aligned with the user's precision and goals.

Directional Stimulus Prompting (DSP) Li et al. (2023) introduced a prompt engineering technique of directional stimulus prompting. The approach involves providing hints in the form of the desired structure or guidance, which have their roles to play in directing the language model towards producing the desired outcome.

For instance, if a downstream task is to generate test case documents, the hint could be specifying the right document outline. The prompt could include guidelines such as 'The test case document should have the following sections: test case title, description, steps, expected results, and notes.' By doing this, the model could familiarize itself with the way the test case document is organized, including the logic structure and used notations.

Through this directional and stimulus guidance, this prompt provides a way of organizing the model's output to ensure that it will consistently remain in alignment with the set purpose or criteria. Giving such crucial signals in the prompt makes the prompt engineer able to align the model with the desired content and setting.

Consequently, the model creates outputs that suit the required formatting or structure and can even give task-specific instructions. Lastly, directional-stimulus prompting is a useful tool in promoting the generation of output that simultaneously meets the user's satisfaction and objectives.

5.3.5 Mistral-7B

Mistral-7B is a causal language model consisting of a transformer decoder block only. It is a cutting-edge language model since it embraces the transformer-based architecture while successfully delivering both robust performance and efficiency while maintaining the balance model property during the rapidly evolved NLP domain (Jiang et al., 2023b). In this section, we dive into its architectural details. However, before we can understand the model architecture, we need to cover a few techniques:

Grouped-query attention (GQA) Ainslie et al. (2023) introduced a generalization of *multi-query attention (MQA)* [6] where an intermediate number of key-value heads is used, still achieving quality close to *multi-head attention (MHA)* in a comparable speed value to MQA (see Figure 5.8).

GQA divides query heads into subgroups, where each subgroup shares a single key head and value head, interposing between MHA and MQA (see Figure 5.9). Thus, it helps keep the same proportional decrease in bandwidth and capacity as model size increases (Ainslie et al., 2023).

Additionally, there is no optimal number for the number of groups, since it could be anywhere between more than one and less than the number of query heads. However, Ainslie et al. (2023) commented on this and concluded that increasing the number of groups from MQA only results in modest slowdowns initially, with increasing cost as we move closer to MHA.

Moreover, due to encoder representations being computed in parallel, memory bandwidth is not the primary bottleneck. Ainslie et al. (2023) pointed out that GQA is not applied to the encoder self-attention layers.

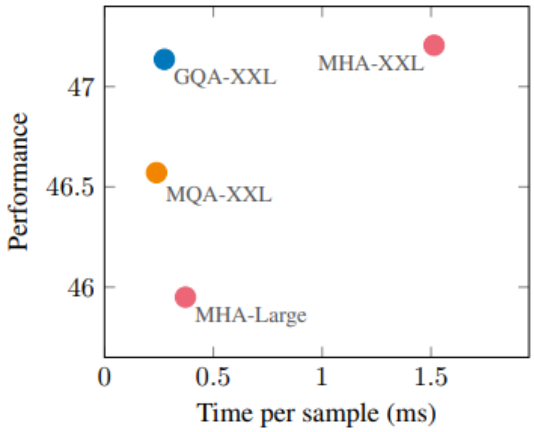

Figure 5.8: Uptrained MQA offers better tradeoff than MHA-Large with improved quality and speed, while GQA is even better than MHA-XXL with comparable quality and similar speed gains (Ainslie et al., 2023).

Sliding window attention (SWA) This method draws advantage from the stacked layers of a transformer to attend information wider than the window size, hence, it still allows one token to watch tokens outside the window. In addition, it reduces the number of dot products to perform, and thus, the performance during training and inference (Jiang et al., 2023b).

Since some "interactions" between the tokens will not be captured, SWA may lead to degradation in the performance of the model. However, focusing on the local context, related to

[6] Shazeer (2019) explained that MQA significantly reduced the size of the tensors thus the memory bandwidth requirements of incremental decoding, through sharing the keys and values across all the various attention heads "queries heads".

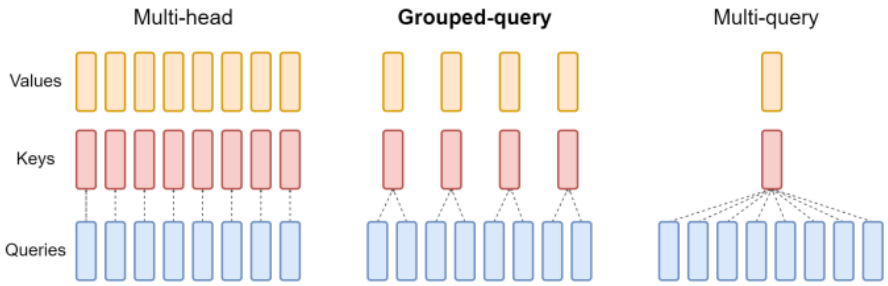

Figure 5.9: Overview of all three attention methods: MHA, GQA, and MQA (Ainslie et al., 2023).

the size of the window, is enough for most cases. If we consider a book, the words used in a paragraph of a chapter are related to other paragraphs of the same chapter, but may not be relevant to chapter 1.

To provide further explanation, we have the following sentence, *"The cat sat on the"* as seen in Figure 5.10. Since we are building an autoregressive model, we begin by using causal attention (also known as vanilla attention), which generates the first matrix on the left. However, it is worth noting that in this matrix, the word *"the"* (last row) has attended to the entire sentence. This is where SWA comes into play. We can address this issue by applying SWA with a window size of 3. Now, the word *"the"* can only attend itself and 2 previous words, namely *"on"* and *"sat"*.

Moreover, with a sliding window size $W = 3$, every layer adds information about $(W-1) = 2$ tokens, which indicates that after N layers, we will have an information flow in the order of $W \times N$.

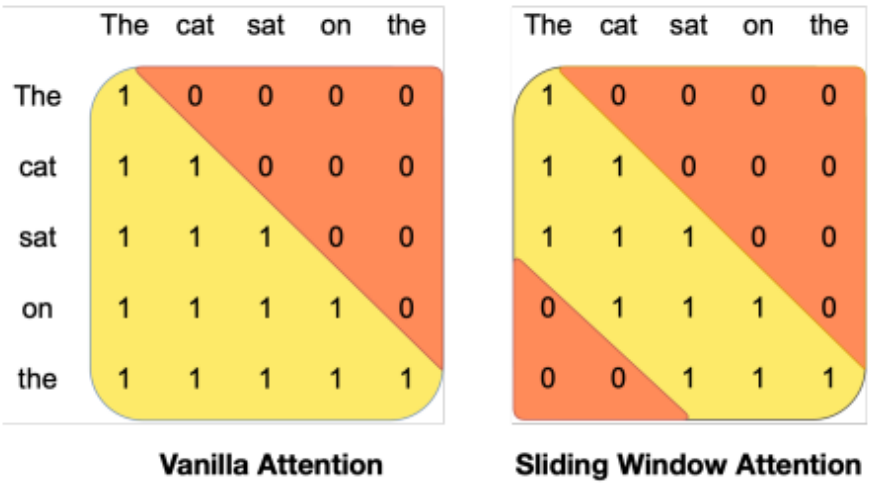

Figure 5.10: Vanilla attention followed by a sliding window attention with a window size of 3 (Jiang et al., 2023b).

Rolling Buffer Cache This method limits the cache size using a rolling buffer cache. Since for each time step i, the corresponding keys and values are stored in a position i

mod W of the cache, where W is the fixed size of the cache, then when the position I is larger than W, all past values in the cache are overwritten to stop the cache from increasing (Jiang et al., 2023b).

Sigmoid Linear Unit (SiLU) is an activation function used in neural networks.

$$\text{SiLU}(x) = x \cdot \sigma(x) \tag{5.3}$$

5.3: Sigmoid Linear Unit function, where $\sigma(x)$ represents the sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$

The SiLU function has been adopted in various deep-learning architectures, including transformer models. Due to its smoothness, it helps in gradient-based optimization during training, thus faster convergence and potentially better performance compared to other activation functions such as *Rectified Linear Unit (ReLU)* (see Figure 5.11) or *Sigmoid* (Elfwing et al., 2018).

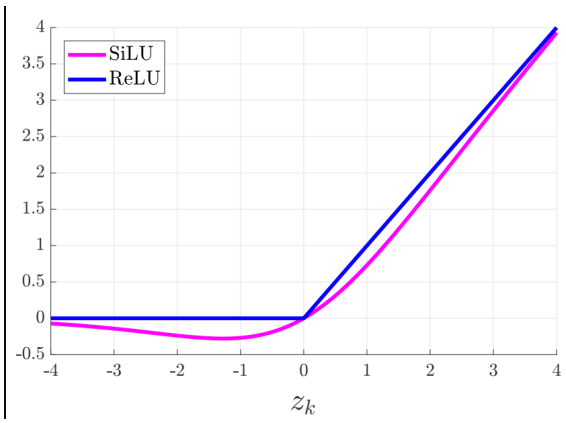

Figure 5.11: The activation functions of the SiLU and the ReLU where z_k is the input to hidden unit k (Elfwing et al., 2018).

Architectural details As seen from Table 5.2, Mistral-7B has 32 encoder layers [7] where each head is attending to 128 dimensions, then the hidden dimension size is 14336 which we can derive simply from multiplying the embedding vector dimension with a factor of 3.5, i.e. $\frac{14336}{4096} = 3.5$.

The number of query attention heads is 32 while the number of heads for key and value is 8. In addition, the inequality between both comes from GQA, and we can note that the number of query groups here is 8, i.e., every 4 query attention heads ($\frac{32}{8} = 4$) have a single key and value head.

Moreover, it is worth naming that Mistral-7B uses **Rotary Positional Encodings** on attention heads for both query and key, since it introduces rotational transformations to the positional encoding vectors, allowing the model to learn richer representations of token positions.

[7] The reason behind the name "encoder layers" is because they are structured like transformer encoder layers which do not have linear and softmax layers, however here in Mistral-7B the normalization layers come before the attention and the feed-forward layers.

Next, the window of Mistral-7B has a fixed size $W = 4096$ for the attention calculation and the rolling buffer cache. Additionally, the model was trained upon a context length of 8192 tokens/words, determining the size of the input sequence that the model can effectively process.

Also, Mistral-7B has a vocabulary size of 32,000, which indicates the total number of unique tokens that the model can recognize and generate during training and inference [8].

As a final note, Mistral-7B does not deploy **Sparse Mixture of Experts (SMoE)** which Pham et al. (2024) explained as a variation of the mixture of experts architecture where only a subset of the available experts are activated or used for each input instance. This sparsity constraint helps reduce computational costs and memory requirements while still allowing the model to benefit from the diversity of expert opinions.

```
<bound method Module.state_dict of MistralForCausalLM(
  (model): MistralModel(
    (embed_tokens): Embedding(32000, 4096)
    (layers): ModuleList(
      (0-31): 32 x MistralDecoderLayer(
        (self_attn): MistralSdpaAttention(
          (q_proj): Linear4bit(in_features=4096, out_features=4096, bias=False)
          (k_proj): Linear4bit(in_features=4096, out_features=1024, bias=False)
          (v_proj): Linear4bit(in_features=4096, out_features=1024, bias=False)
          (o_proj): Linear4bit(in_features=4096, out_features=4096, bias=False)
          (rotary_emb): MistralRotaryEmbedding()
        )
        (mlp): MistralMLP(
          (gate_proj): Linear4bit(in_features=4096, out_features=14336, bias=False)
          (up_proj): Linear4bit(in_features=4096, out_features=14336, bias=False)
          (down_proj): Linear4bit(in_features=14336, out_features=4096, bias=False)
          (act_fn): SiLU()
        )
        (input_layernorm): MistralRMSNorm()
        (post_attention_layernorm): MistralRMSNorm()
      )
    )
    (norm): MistralRMSNorm()
  )
  (lm_head): Linear(in_features=4096, out_features=32000, bias=False)
)>
```

Figure 5.12: Overview of Mistral-7B architecture gained using state_dict function.

[8] Inference is the deployment phase of the machine learning pipeline, where the model applies what it has learned during training to real-world data (Wachter and Mittelstadt, 2019).

Table 5.2: Mistral-7B structure based on Figure 5.12.

Parameter	Description	Value
dim	Size of the embedding vector	4096
n_layers	Number of encoder layers	32
head_dim	dim / n_heads	128
hidden_dim	Hidden dimension in the feedforward layer	14336
n_heads	Number of attention heads (Q)	32
n_kv_heads	Number of attention heads (K, V)	8
window_size	sliding window size for the attention calculation and Rolling Cache	4096
context_len	Context on which model was trained upon	8192
vocab_size	Number of tokens in the vocabulary	32000
num_experts_per_tok	Number of expert models for each token	N / A
num_experts	Number of expert models	N / A

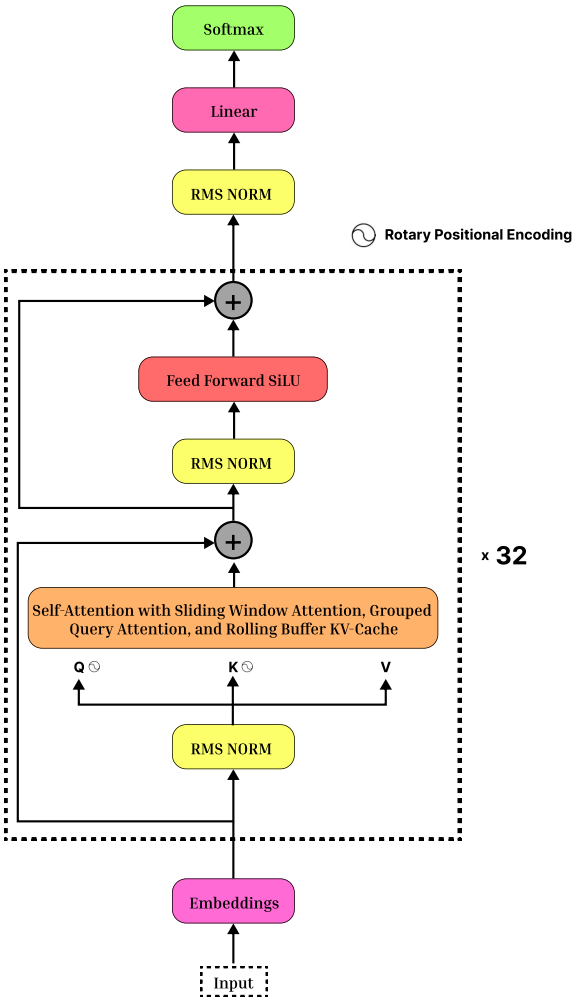

Figure 5.13: Overview of Mistral-7B architecture.

5.4 Llama-cpp-python Integration

The main goal of the llama-cpp library is described by Lin et al. (2023) as to enable LLM inference with minimal setup and state-of-the-art performance on a wide variety of hardware locally and in the cloud, in pure C/C++. Additionally, to leverage this library in this thesis, we used the Python bindings for llama-cpp (Lambert et al., 2010) known as llama-cpp-python.

llama-cpp-python supports different techniques such as:

Chat completion helps the model format the messages into a single prompt using pre-registered chat formats such as *llama-2*, or by providing a custom chat handler object. This includes adding the beginning of sentence token **<s>**, the start of the instruction **[INST]**, and the end of the instruction **[/INST]**.

Additionally, llama-cpp-python allows creating chat completion during inference using *create_chat_completion* function. It generates a chat completion from a list of messages, with a predefined temperature [9], a max number of new tokens to generate, a penalty for repeated tokens, and a response format such as JSON object.

Speculative decoding is defined by Leviathan et al. (2023) as an algorithm to sample from autoregressive models faster without any output changes, by computing several tokens in parallel. This method makes the inference phase faster since transformers decoding M tokens take M serial runs of the model.

First, providing a small and cheap draft model whose goal is to generate a candidate sequence of M tokens (a draft). Next, feeding all these tokens to the target (big) model in a batch is as fast as feeding in one token (see Figure 5.14).

Thereafter, the big model goes through the sequence and sample tokens. Samples that agree with the draft allow the model to immediately skip forward to the next token. However, the samples that do not match the draft, make the model throw the draft away and take the cost of throwing. The key here is that the draft tokens get accepted most of the time because even a small draft model can predict them. While the hard tokens where the big model disagrees fall back to the original speed (Leviathan et al., 2023).

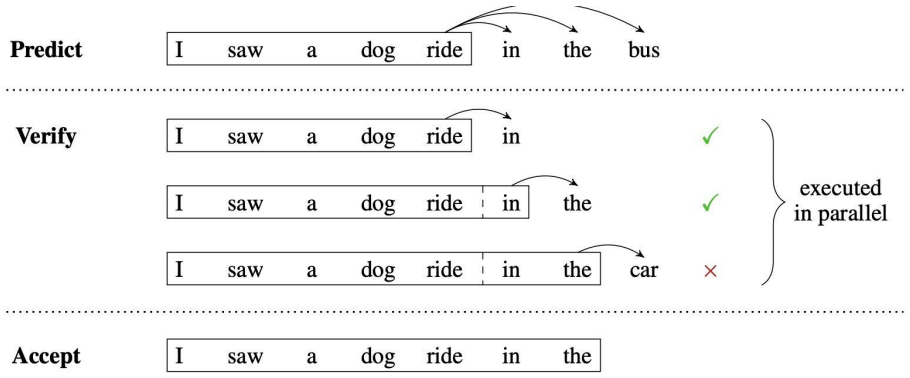

Figure 5.14: Speculative decoding method (Stern et al., 2018).

[9] Temperature is a hyperparameter that controls the level of randomness or creativity in the generated text (Renze and Guven, 2024).

Supporting GPT-Generated Unified Format (GGUF) models GGUF is described by Haaralahti (2024) as a quantization method that allows using of the CPU to run an LLM but also offload some of its layers to the GPU for a speed-up (see Table 5.3). However, in llama-cpp-python, it is feasible to offload all the layers to the GPU using the parameter *n_gpu_layers = −1*, which shortens the inference time.

Table 5.3: Quantized models of Mistral-7B-Instruct-v0.2 with GGUF (Hong et al., 2017). RAM figures below assume no GPU offloading. If layers are offloaded to the GPU, this will reduce RAM usage and use VRAM instead.

Model Name	Quant Method	Bits	Max RAM Required	Information
mistral-7b-instruct-v0.2.Q5_K_M.gguf	Q_5_K_M	5	7.63 GB	Uses Q6_K for half of the attention.wv and feed_forward.w2 tensors, else Q5_K. Recommended as it preserves most of the model's performance.
mistral-7b-instruct-v0.2.Q8_0.gguf	Q_8_0	8	10.20 GB	is almost indistinguishable from float16. High resource use and slow.

Chapter 6

Experiments

This chapter describes the methodology of the proposed approach as well as the experiments done in this thesis. An overview of the approach is described, to begin with, then it is followed by a section where the setup and deployment are defined. Subsequently, the next section focuses on the evaluation methodology and the chosen metrics to measure the performance of each experiment.

6.1 Overview of the Approach

The goal of this thesis is to design, implement, and evaluate an approach that automates the process of test case generation based on feature descriptions written in natural language. In addition, the proposed approach is implemented as a tool that from a given unseen feature description generates suggestions of natural language test cases that could be used for testing different features.

The approach leverages NLP techniques to generate test cases written in natural language and following specific guidelines, given feature descriptions written in natural language as well. First, we parse the feature descriptions to use them later while formatting the prompt. Then, we define a format prompt function to apply to the parsed feature descriptions to create the input. The created input is then given to the chosen model to generate suggested test cases, which are then parsed to get clear test cases written in natural language (see Figure 6.1).

6.2 Setup & Deployment

We used Python to implement the proposed approach due to its popularity and its useful machine-learning packages. In this section, we present the tools we used in the study.

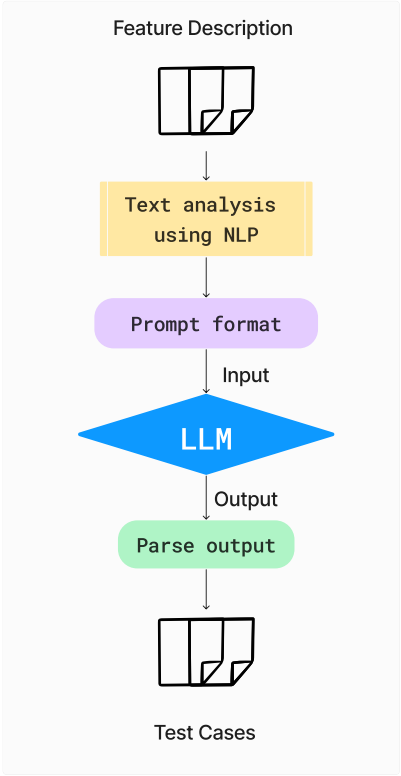

Figure 6.1: The stages of the proposed approach

6.2.1 Choice of Model

To predict the next token in a sequence, we need a good autoregressive language model, which takes into account both the previous tokens and the token itself, thus we chose both Llama-7b (Touvron et al., 2023), and Mistral-7b (Jiang et al., 2023b). However, Mistral-7b outperforms Llama 2 13b on all benchmarks (Lunney et al., 2006), thus, we chose Mistral 7b as our LLM in this study.

Thereafter, we decided to examine the instruction version of Mistral-7b, *Mistral-7b-Instruct-v0.2*, which illustrates the model's generalization capabilities through fine-tuning on instruction datasets. Jiang et al. (2023a) discussed that Mistral-7b-Instruct outperforms all other 7B models on MT-Bench and competes well with 13B chat models (see Figure 6.2). Subsequently, we chose Mistral-7b-Instruct-v0.2 as our selected LLM.

6.2.2 Defining Setup

It is crucial to establish a default setup that ensures consistency and reproducibility in training our selected LLM. In this stage, we define a default setup for all experiments to streamline the training workflow.

Model Deployment on Single GPU In this part of the study, the direction turned to applying the model for fine-tuning purposes on a single GPU. The only available hardware for this study was an RTX 4080 16GB GPU.

Model	MT Bench
WizardLM-13b-v1.2	7.2
Vicuna-13B-16k	6.92
Mistral 7B Instruct	**6.84 ± 0.065**
WizardLM-13B-v1.1	6.76
Llama-2-13b-chat	6.65
Llama-2-7b-chat	6.27
Vicuna-7B-16k	6.22
Alpaca-13B	4.53

Figure 6.2: Mistral-7B-Instruct achieves a score of 6.84 on the MT-Bench. As demonstrated, this score beats models like Llama 2 13B chat, and Llama 2 7Bchat (Minaee et al., 2024).

To be able to load and fine-tune the model effectively on the allocated GPU, it was necessary to resolve memory constraints. Initially considering 8-bit quantization with QLoRA, it became clear that this approach did not give enough memory headroom for fine-tuning operations. As a result, we decided to go for a more targeted approach and implement 4-bit quantization.

After applying QLoRA with 4-bit quantization, the model experienced significant compression while still maintaining all the required parameters for effective fine-tuning. The post-quantization statistics of the model are shown in Table 6.1:

Table 6.1: Number of trainable parameters after QLoRA

Instance	Number of Parameters
Total trainable parameters for Mistral 7B Instruct	7,284,252,672
Trainable parameters after 4-bit Quantization with QLoRA	42,520,576
Trainable parameter ratio	0.58373%

These statistics demonstrate how well quantization works in reducing the memory footprint of the model and how in combination with PEFT enables fine-tuning tasks on a single consumer GPU. With the combined use of advanced techniques like QLoRA and PEFT, we were able to find a good balance between computational efficiency and model performance, making efficient fine-tuning procedures possible under our circumstances.

Defining default training arguments involves specifying the standard configurations and parameters used during the training process. These arguments cover different aspects such as *save strategy*, *evaluation strategy*, *training batch size*, *evaluation batch size*, *number of epochs*, *logging steps* etc. Additionally, default training arguments serve as a starting point for customization, allowing us to optimize the parameters later, based on specific requirements and knowledge from observations.

Defining default hyperparameters provides predefined values for a set of chosen hyperparameters to initialize an experiment before optimizing the values (see Tables 6.2 and 6.3).

Table 6.2: A set of chosen hyperparameters with their corresponding predefined values for fine-tuning approach

Hyperparameter	Predefined Value
LoRA Rank	16
LoRA Dropout	0.1
LoRA Alpha	16
Noise Embedding Alpha	0
Optimizer	paged_adamw_32bit
Learning Rate	2×10^{-4}
Learning Scheduler Type	cosine
Weight Decay	0.001
Warmup Ratio	0.05

Table 6.3: A set of chosen hyperparameters with their corresponding predefined values for directional stimulus prompting approach

Hyperparameter	Predefined Value
Temperature	0.8
Max Tokens	5012
Number of Predicted Tokens	20
top_p	0.95
top_k	40

Defining callback functions which also serve as powerful tools for monitoring and controlling the training process. We used two callback functions, one to save the checkpoint of the model and the other one to stop training when the specified metric worsens for an integer called patience evaluation calls.

Listing 6.1: Callback functions

```
class PeftSavingCallback(TrainerCallback):
    def on_save(self, args, state, control, **kwargs):
        checkpoint_path = os.path.join(args.output_dir,
            f"checkpoint-{state.global_step}")
        kwargs["model"].save_pretrained(checkpoint_path)

        if "pytorch_model.bin" in os.listdir(checkpoint_path):
            os.remove(os.path.join(checkpoint_path,
                "pytorch_model.bin"))

callbacks = [PeftSavingCallback,
    EarlyStoppingCallback(early_stopping_patience=2)]
```

Choosing and loading the trainer implies selecting the appropriate trainer with its arguments to prepare the model for the training process. We used the SFT trainer with chosen datasets for both training and validation, together with a predefined PEFT configuration, and a tokenizer.

Listing 6.2: SFT trainer with its arguments

```
trainer = SFTTrainer(
    model=model,
    train_dataset=train_data,
    eval_dataset=eval_data,
    peft_config=peft_config,
    dataset_text_field="text",
    tokenizer=tokenizer,
    args=training_arguments,
    callbacks=callbacks,
    packing=False,
    max_seq_length=4096)
```

6.3 Running Experiments Strategies

To run different experiments, we applied various strategies to efficiently analyze and evaluate each experiment. These strategies include three techniques, fine-tuning, prompt engineering, and an AI agentic workflow. Additionally, we used experiment tracking and logging tools to monitor and record the results of each experiment, enabling thorough analysis and comparison of model performance under different conditions.

We did the monitoring experiments through a development platform, *Weights & Biases* (Stiny, 1992), that allows users to gain insights into model behavior, visualize, and track in real-time different aspects of the model training process.

6.3.1 Prompt engineering approach

In this section, we will touch on the different experiments we conducted using diverse prompt engineering techniques. We initially started by applying prompt engineering with different shots. We proceeded with our prompt engineering approach by applying the prompt chaining technique and finally we utilized the directional stimulus technique with the deployment of llama-cpp-python.

Prompting experiments with different shots

Zero-shot: In this prompt, we did not provide any instructions nor additional information or context besides what we wanted the model to do, see the code in Listing 6.3 below:

Listing 6.3: Zero-shot prompt template function

```
def zero_shot_format(name_of_the_feature):
    return f"""
    <s>[INST]
        Generate test cases for this feature:
            {name_of_the_feature}
    [/INST] \n""".strip()
```

Half-shot: We described the model's supposed behavior and provided the input document with specific instructions defining the expected structure of the output. The structure in our case is the different sections of the test document and their order, see Listing 6.4.

Listing 6.4: Half-shot prompt template function

```
def half_shot_format(datapoint):
        return f"""<s>[INST]
        You are an AI software testing assistant who is going
            to generate test case documents based on the
            specification documents given in the square
            brackets
        [{datapoint['documentations']}]
        Each test case document should have the following
            structure and subsections:
        Test Case Title \n
        Objective \n
        Approval Criteria \n
        Limitations \n
        Execution \n
        [/INST] \n""".strip()
```

One-shot: The one-shot prompt had the same structure and contents as the half-shot prompt but was appended with an example of an input and output, see Listing 6.5.

Listing 6.5: One-shot prompt template function

```
def one_shot_format(datapoint):
        return f"""<s>[INST]
        You are an AI software testing assistant who is going
            to generate test case documents based on the
            specification documents given in the square
            brackets
        [{datapoint['documentations']}]
        Each test case document should have the following
            structure and subsections:
        Test Case Title \n
        Objective \n
        Approval Criteria \n
        Limitations \n
        Execution \n

        Here is an example of a feature specification with
            its corresponding test cases:
        specification: {example['documentation']}
        test cases: {example['testcases']}
        [/INST] \n""".strip()
```

Chain prompting experiment

In our prompt engineering approach, one of the experiments we conducted utilized the chain prompting technique to guide the model into generating higher-quality test cases. We formatted around a dozen prompts to be used in prompt chains. With our constructed prompts, we created different combinations and lengths of prompt chains which were fed into the model to find the most optimal length and combination of prompts.

The prompts were constructed to guide the model with feedback on the previously generated test cases. The prompts would include guiding instructions informing the model of what it should improve in the next generated test cases, see Listing 6.6.

Listing 6.6: Example of a guiding prompting in the chain

```
guiding_prompt = f"""The previous test case documents were not
    relevant enough to the feature specification document I
    provided above.
Rewrite the test case documents and make them heavily based on the
    section <How to use> from the specification document I provided.
The test case documents should still have this structure and the
    following subsections:
    Test Case Title \n
    Objective \n
    Approval Criteria \n
    Limitations \n
    Execution \n"""
```

Llama-cpp-python prompting experiments

In this experiment, we applied the directional stimulus prompting technique to examine if it improves the model output. Moreover, as we discussed the benefits of using llama-cpp in the architecture chapter, we decided to leverage them with directional stimulus prompting.

First, we downloaded from *Hugging Face* two models in GGUF format, mistral-7b-instruct-v0.2.Q5_k_M and mistral-7b-instruct-v0.2.Q8_0 (see Table 5.3). Then, we used *llama-2* as the chat format, since it includes the same special tokens our model uses. Thereafter, we defined a list of messages to use in the chat completion function and guided the model with a specific response format, as seen below.

Listing 6.7: List of guided messages with half-shot approach

```
messages=[
            {
            "role": "system",
            "content": """You are an AI software testing assistant who
                is going to generate test cases based on the
                specification documents.
                       The test cases you generate should be heavily
                           based on the <How To Use> section in the
                           specification document.
                       Generate more than one test case if needed.
```

```
                Each test case document should have the
                    following structure and subsections:\n
                Test Case Title \n
                Objective \n
                Approval Criteria \n
                Limitations \n
                Execution \n""",
        },
        {"role": "user", "content": row['documentations']}
```

Moreover, we selected a collection of hyperparameters (refer to Table 6.3) that exert the most significant influence on the model (see Table 6.4). In addition, we relied on *LlamaPromptLookupDecoding* (Saxena, 2023) as a draft model for speculative decoding.

Listing 6.8: Guiding the model with a specific response format

```
response_format={
    "type": "json_list",
    "schema": {
        "type": "object",
        "properties": {
            "test_case_title": {"type": "string"},
            "objective": {"type": "string"},
            "approval_criteria": {"type": "string"},
            "limitations": {"type": "string"},
            "execution": {"type": "string"},
        },
        "required": ["test_case_title", "objective",
            "approval_criteria", "limitations", "execution"],
    },
}
```

Table 6.4: A set of chosen hyperparameters with their corresponding effect for llama-cpp prompting experiments

Hyperparameter	Effect
Temperature	Controls randomness, higher values increase diversity.
Max Tokens	The maximum number of tokens to generate.
Number of Predicted Tokens	Number of predicted tokens for the draft model.
top_p	The cumulative probability cutoff for token selection. Lower values mean sampling from a smaller, more top-weighted nucleus (Holtzman et al., 2020).
top_k	Sample from the k most likely next tokens at each step. Lower k focuses on higher probability tokens (Sinha et al., 2020).

6.3.2 Fine-tuning approach

In this approach, we used our pre-trained model. We leveraged the knowledge encoded in it, followed by fine-tuning it on our predefined datasets to update its weights on our downstream task which is generating test cases. Moreover, we wanted to experiment with fine-tuning in combination with prompt engineering and with/without HPO.

Fine-tuning with half-shot

Here, we applied the aforementioned half-shot prompt function to all training and validation data, then we fine-tuned the model using this data.

Listing 6.9: Applying half-shot prompt on both training and validation data before fine-tuning

```
X_train = pd.DataFrame(X_train.apply(half_shot_format, axis=1),
    columns=['text'])
X_eval = pd.DataFrame(X_eval.apply(half_shot_format, axis=1),
    columns=['text'])
```

Fine-tuning with default parameters involves fine-tuning with minimal adjustments to the default parameters. This approach is often used when the downstream task is akin to the task the model was originally trained on.

The default parameters mentioned in Table 6.2 are used for this method. Furthermore, we will define their effects to gain insight into how each one affects the model (see Table 6.5).

Fine-tuning with HPO aims to systematically explore the hyperparameter with a search space, using different HPO techniques. By running different trials, with various values for the predefined hyperparameters, one could choose the trial with the best evaluation metric result. For this approach, we used NNI as an HPO platform.

Table 6.5: A set of chosen hyperparameters with their corresponding effect

Hyperparameter	Effect
LoRA Rank	determines the rank of low-rank approximation used in LoRA. Higher ranks demand better GPUs (more VRAM) but may capture more complex patterns in the data.
LoRA Dropout	refers to dropout probability used in LoRA. It helps prevent overfitting.
LoRA Alpha	regularization strength parameter for LoRA. Higher values result in increasing the penalty on large weights, thus controlling model complexity and preventing overfitting.
Noise Embedding Alpha	As Jain et al. (2023) discussed, noisy embedding instruction fine-tuning alpha adds noise to the embedding vectors during training. This has been proven to improve performance on modern instruction datasets. It has been proved by Jain et al. (2023) that LLaMA-2 Chat benefits from additional training with NEFTune.
Optimizer	determines the update rule used to apply weight adjustments during training. Different optimizers may perform differently based on their properties. Here, as the initial optimizer, we chose paged_adamw_32bit, a variant of AdamW optimizer designed specifically to train LLMs and help with limited memory resources.
Learning Rate	interprets, during optimization, the step size taken. Higher values may lead to converge faster, but may also result in instability or overshooting.
Learning Scheduler Type	refers to how the learning rate changes throughout training. It can affect convergence speed.
Weight Decay	involves adding a penalty term to the loss function for large weights. It helps prevent overfitting.
Warmup Ratio	determines the relative importance of positive and negative examples in the loss calculation.

6.3.3 AI Agentic Workflow Approach

Since LLMs have recently shown promising outcomes in decision-making and planning capabilities compared to humans, LLM-based agents, as Guo et al. (2024) described, have been rapidly developed to understand and generate humanlike instructions.

In addition to the escalating advancement of AI agents, a more iterative approach, known as agentic workflow, has been widely employed to execute tasks by utilizing these agents in combination with LLMs. Moreover, different collaborative working systems, such as **Crew AI** (Hassan et al., 2024), come into play, allowing teams of AI agents to efficiently work together to accomplish various complex tasks.

In this model, each team member operates as an agent, empowered to make decisions and take actions aligned with the overall goals of the task. Each agent has predefined skills and a particular job, along with search tools that can be provided (see Figure 6.3), enabling everything from simple searches to complex interactions and effective teamwork among agents (Hassan et al., 2024).

In this thesis, we used Crew AI to build an AI agentic workflow with a team (crew)

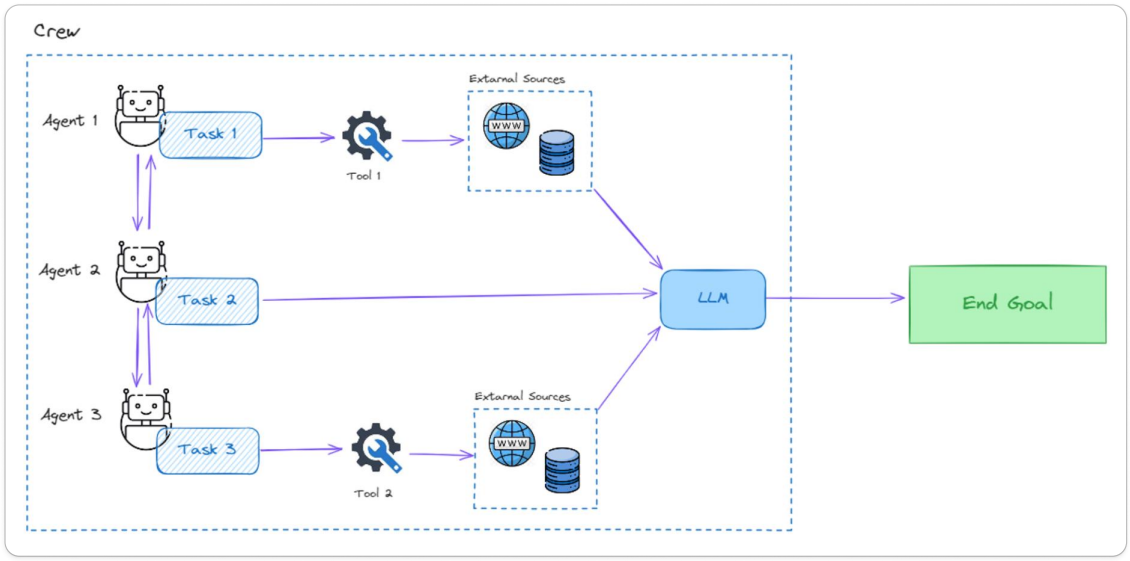

Figure 6.3: Overview of the agentic workflow model with three AI agents with different tasks and tools, working with an LLM to accomplish an end goal (Guo et al., 2024).

consisting of four agents with different skills, roles, and tasks as the following:

Expert Interpreter Has the role of feature document interpreter, where its goal is to analyze the provided feature, extract all relevant details, and compose a comprehensive description to use in writing the test cases.

Test Case Writer Has the objective of generating test cases based on the provided feature documentation. It delivers multiple test cases with the desired structure.

Test Case Evaluator Its goal is to assess the quality of the generated test cases by the test case writer. In addition, it should provide constructive feedback for improvement.

Final Test Case Writer Is an expert in writing final test case documents. It compiles and formats the final version of the test case document based on the evaluated test cases. This agent refines the document structure, ensures consistency, and produces a polished and ready-to-use test case document.

After that, we chose a sequential task execution for the crew. Moreover, regarding the model, Crew AI allows integration of local LLMs only through **Ollama**[1] since it is preferred for local LLM integration, offering customization and privacy benefits (Hassan et al., 2024). Hence, we downloaded Ollama and pulled Mistral 7B to run it locally.

Subsequently, we implemented a pipeline to automate the use of the crew by first generating test cases for each sample in the test dataset, and then evaluating each generated sample. It is noteworthy to mention that Crew AI permits each agent to be represented and operated by an LLM, with the default being "GPT-4" if no other LLM is specified. However, in this thesis, we chose Mistral 7B pulled from Ollama for all agents.

[1] `https://github.com/ollama/ollama`

Listing 6.10: Defining the crew.

```
crew = Crew(
        agents=[expert_interpreter, test_case_writer,
            test_case_evaluator, final_test_case_writer],
        tasks=[expert_interpreter_task, test_case_writer_task,
            test_case_evaluator_task, final_test_case_writer_task],
        verbose=2,
        process=Process.sequential,
        max_rpm=10,
        share_crew=False
    )
```

Approach Limitations

During the agentic workflow approach, we had different restrictions that tied us. We could not enable the memory parameter, a feature that comprises short-term memory, long-term memory, entity memory, and newly identified contextual memory, each serving a unique purpose in aiding agents to remember, reason, and learn from past interactions (Hassan et al., 2024) for the crew due to limited GPU.

Moreover, we encountered challenges in facilitating delegation among the agents, a feature that empowers them to assign tasks or queries to one another. This functionality ensures that the most appropriate agent addresses each task. However, we faced frequent connection loss errors, potentially linked to the company's infrastructure. However, these errors did not occur when running Crew AI from home.

6.4 Evaluation

Evaluation is a crucial aspect of machine learning, providing insights into the performance and robustness of the employed models. By evaluating, the quality of the model output can be evaluated against some predefined criteria.

In natural language generation (NLG), a sub-field of NLP is associated with building software systems that can produce coherent and readable text (Reiter and Dale, 1997), the evaluation is challenging since NLG tasks are open-ended. For instance, for the same input, the NLG model can generate multiple appropriate responses (Celikyilmaz et al., 2021).

Therefore, when it comes to most NLG tasks, human evaluation remains the superior standard. However, it is expensive, time-consuming, and labor-intensive. Hence, NLG evaluation approaches developed recently serve a critical role in guiding system development and ensuring that generated text fulfills the desired quality standards for its intended use case.

6.4.1 NLG Evaluation Metrics

Common evaluation metrics such as training loss and validation loss provide valuable insights into a model's performance and efficiency. However, they are not always suitable for NLG tasks because they are open-ended. Unlike tasks like classification, NLG involves generating text, which poses challenges for traditional metrics.

Additionally, supervised training generally concentrates on minimizing a loss function such as cross-entropy. However, in many applications, we are interested in performing well on metrics that pertain to the application (Song et al., 2016). Hence, appropriate evaluation metrics ensure that the model meets the required objectives and produces reliable results in real-world applications.

BLEU stands for Bilingual Evaluation Understudy (Papineni et al., 2002) and is a metric that is used to measure how good a machine translation using human reference translations. It evaluates the accuracy of n-grams of varying lengths (from 1 to 4 word) in the candidate against the reference translations. The brevity penalty is introduced into BLEU to avoid giving high scores to overly short translations. The final score is an integer between 0 and 1 where the higher the number the closer the candidate translations are to the reference translations. BLEU gets used very often but has also been criticized for not taking meaning or context into account, and too often assigns high scores to low quality translations.

ROUGE stands for Recall-Oriented Understudy for Gisting Evaluation (Lin, 2004) and metric for evaluation of the quality of summaries or machine translation in comparison to one or more reference text. It computes coverage of n-grams, sequences of words or word pairs in candidate and reference documents. There are key variants of ROUGE: ROUGE-N – the N-gram overlap-based approach; ROUGE-L – the Longest common subsequence to consider for word order. ROUGE focuses mostly on recall because it measures the percentage of the reference text contained in the candidate text; however, ROUGE can also calculate precision and F1-score. This makes ROUGE useful for evaluating content coverage and coherence in summarization and translation.

METEOR the Metric for Evaluation of Translation with Explicit Ordering (Lavie et al., 2004). It evaluates machine translation by using correlations between candidate translation and reference translation with precision and recall. It uses Word Alignment for aligning words between candidate and reference texts and for identifying variations through stemming and synonymy matching. F1 scores are generated by METEOR, which is a harmonic mean of precision and recall. It also poses a penalty on word order variations to correct for fluency and coherence.

Similarity scores In the context of text generation, a similarity score measures how similar or related a generated text is to a given reference text or set of texts. Two popular scores were used in this thesis, Levenshtein distance, Jaccard, and spaCy. The first one is also known as the edit distance, it measures the minimum number of single-character edits (insertions, deletions) required to change one sequence into another (Celikyilmaz et al., 2021). Jaccard's similarity, on the other hand, calculates the degree to which something is similar by dividing its intersection by its union. Furthermore, spaCy doc similarity (Zhelezniak et al., 2019) uses an average of word vectors to estimate semantic similarity based on cosine similarity.

6.4.2 Human Evaluation

Involving humans as judges, as Celikyilmaz et al. (2021) discussed, is the most natural way of evaluating a system. In this study, we asked humans of various roles (senior, junior, and experienced engineers) to rate different test cases.

Our human evaluation approach was based on eight different samples, where six of them belong to already existing features while the remaining two correspond to completely new features that do not have written test cases yet. Additionally, for the already existing features, we mixed real (human-written) test cases with the generated test cases gained from our models to make it a bit challenging for the judges, encouraging them to analyze the given test cases properly.

Thereafter, for each test case, we asked the judges to set a score based on the following criteria:

$$\text{Score} \in [1, 10], \quad \text{where} \begin{cases} 1 & \text{means irrelevant test case/obviously machine-written,} \\ 5 & \text{means human-written test case/valid test case,} \\ > 5 & \text{means very accurate and much better than a human-written version.} \end{cases}$$

6.4.3 Evaluation Pipeline

To measure all aforementioned evaluation metrics, we built a pipeline to get a closer look at the model performance (see Figure 6.4). It involves several stages, starting with generating test cases (candidates) for the entire test dataset (76 samples), and then extracting all real test cases (references) for each sample in the same dataset.

Next, to resolve the problem of references and candidates that may have different lengths in terms of the number of test cases, we had to apply padding or flattening depending on the used metric and which technique it benefits most.

For padding, we explored two distinct methods. Firstly, employing the “copy” approach, which involves appending a randomly selected test case from the same list to the shorter list, aligning its length with the longer list. Secondly, employing a “pad token” strategy, where the shorter list is padded with the chosen pad token **<PAD>** to achieve the same parity in length with the other list.

Thereafter, we evaluated each sample on all six metrics, and then we calculated the average score of the entire test dataset for each metric to get an overall score.

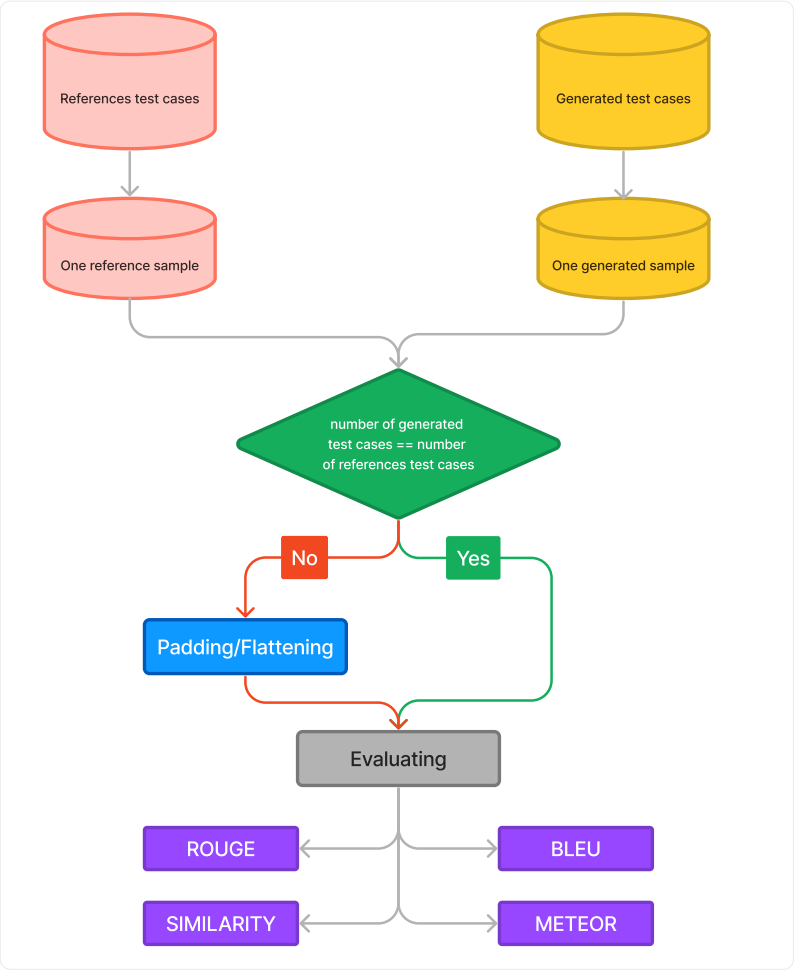

Figure 6.4: The evaluation pipeline we designed to get valuable insights into the model performance.

Chapter 7
Results & Discussion

In this chapter, we report the results of our different experiments. First, we dive into outcomes from fine-tuning the basic model, in which manual QLoRA quantization techniques are involved. Next, we discuss experiments that utilized prompt engineering on the base model and through the llama-cpp-python framework that uses GGUF quantization, aimed at improving the model's performance. Lastly, we present the results from the experiments performed with the agentic workflow approach, determining how effective this method is on our downstream task.

It is important to mention that the contents of the result tables below only represent the highest-performing experiments. Significantly more experiment trials have been conducted, but we will only focus on the main findings.

Our mission is to identify which engineering techniques are most successful and how they are related to the quality of text generated. Finally, we cover the pathways for future study and exploration in this section. In all the tables presented in this chapter, "Matched" indicates the number of samples from the test dataset where the number of generated test cases matches the number of reference test cases for that sample.

7.1 Fine-tuning Results

In this subsection, we present the findings and the results of our fine-tuning experiments. First, the results of fine-tuning the model revealed unexpected outcomes, where it performed much worse than the pre-trained base model which had a BLEU score of 30.18. Despite our efforts to refine the model through fine-tuning on domain-specific data, the model kept getting worse, the more data it got. As demonstrated in Table 7.1, there is a significant drop in performance scores when comparing the model fine-tuned on the small dataset to the one fine-tuned on the medium dataset.

These results raise questions about the efficiency of this approach for the given downstream task. We hypothesize that this is because we built the datasets using the base model

through different data augmentation techniques, due to the lack of data. Hence, we conclude that the fine-tuning process damaged the model weights. Further investigation is recommended to identify the underlying factors contributing to this discrepancy.

Table 7.1: Fine-tuning result table on Mistral 7B Instruct v0.2 with 3 epochs

Dataset	Matched	Loss	BLEU	Meteor	Rouge	Similarity
Small Dataset	17	*Train*: 0.57 *Val*: 0.76	22.48	0.18	*R1*: 0.24 *R2*: 0.05 *RL*: 0.13 *RLsum*: 0.23	*Lev*: 0.22 *Jacc*: 0.71 *spaCy*: 0.8
Medium Dataset	49	*Train*: 0.4 *Val*: 0.34	7.33	0.13	*R1*: 0.13 *R2*: 0.04 *RL*: 0.08 *RLsum*: 0.12	*Lev*: 0.09 *Jacc*: 0.51 *spaCy*: 0.49
Large Dataset	43	*Train*: 0.65 *Val*: 0.76	9.31	0.12	*R1*: 0.18 *R2*: 0.05 *RL*: 0.1 *RLsum*: 0.16	*Lev*: 0.1 *Jacc*: 0.42 *spaCy*: 0.45

7.2 Prompt Engineering Results

Our experiments on prompt engineering offered promising results, proving the efficacy of appropriate prompts in enhancing model performance. Through systematic variation of prompts, we observed significant improvements in the generated test cases. Specifically, cases where prompts were finely tuned to guide the model or where the model hinted by the prompts toward desired outputs showed improved performance compared to unguided approaches. These findings demonstrate the crucial role of prompt design in optimizing model behavior and hold promise for advancing the field of NLP.

7.2.1 Base Model with QLoRA

DSP with few-shots

Our conducted experiments using prompt engineering to apply few-shot learning with DSP techniques implied that the provision of contextual examples leads to the generation of higher-quality text. Table 7.2 below demonstrates the results acquired in different metrics applying a varying number of shots. Thus, we are observing that executing a few shots is helping us to generate more consistent output figures as well as improving the BLEU, Rouge, and Similarity scores.

On the other hand, it is worth mentioning that although increasing the number of shots usually leads to better performance, our results do not necessarily indicate that increasing

the number of shots translates into improved performance. From Table 7.2, we can see that 1-shot achieved the highest scores in all metrics. As the shots were incrementally increased from 1-shot, we found that 2-shots, 3-shots, and 4-shots reduced the scores. Due to hardware resource limitations, we were not able to experiment with 5-shots on the model we quantized with QLoRA, see Table 7.2.

Table 7.2: DSP with few-shot prompt engineering result table on Mistral 7B Instruct v0.2 with QLORA

Shots	Matched	BLEU	Meteor	Rouge	Similarity
Zero-shot	5	4.2	0.138	*R1*: 0.23 *R2*: 0.05 *RL*: 0.15 *RLsum*: 0.0	*Lev*: 0.194 *Jacc*: 0.704 *spaCy*: 0.73
Half-shot	27	19.77	0.13	*R1*: 0.29 *R2*: 0.06 *RL*: 0.16 *RLsum*: 0.27	*Lev*: 0.13 *Jacc*: 0.70 *spaCy*: 0.76
1-shot	**42**	**30.18**	**0.24**	*R1*: **0.30** *R2*: **0.08** *RL*: **0.18** *RLsum*: **0.28**	*Lev*: **0.23** *Jacc*: **0.72** *spaCy*: **0.79**
2-shot	27	29	0.22	*R1*: 0.29 *R2*: 0.07 *RL*: 0.17 *RLsum*: 0.27	*Lev*: 0.22 *Jacc*: 0.71 *spaCy*: 0.77
3-shot	29	29.96	0.22	*R1*: 0.298 *R2*: 0.07 *RL*: 0.17 *RLsum*: 0.27	*Lev*: 0.21 *Jacc*: 0.71 *spaCy*: 0.77
4-shot	27	29.34	0.22	*R1*: 0.284 *R2*: 0.07 *RL*: 0.17 *RLsum*: 0.27	*Lev*: 0.21 *Jacc*: 0.70 *spaCy*: 0.75
5-shot	CUDA	Out	Of	Memory	Error

7.2.2 Model with GGUF through Llama-cpp-python

DSP with few-shot

This experiment looks at the impact of a different number of shots with DSP applied using llama-cpp-python and both 5-bit and 8-bit GGUF quantization on Mistral 7B Instruct.

There is a remarkable trend of better performance as the number of shots increases. It is demonstrated in Table 7.3 that the metrics Matched, BLEU, Rouge, and Similarity improved while increasing the number of shots. Nevertheless, we found that, regarding the 5-bit model, once a point is reached, further increments in the number of shots cause performance to reach a plateau level or even decline. Additionally, for the 8-bit, as Table 7.4 indicates, the half-shot had the best BLEU score, and the more shots we gave the model, the worse the scores. We did hyperparameter optimization on the half-shot model, where we ran 50 trials with a search space for the hyperparameters described in Table 6.4. We gained a slight performance improvement with a BLEU score of **32.925**, the highest achieved score so far.

A worth mentioning observation is the successful implementation of the 5-shot scenario with llama-cpp-python GGUF quantization that proves the efficiency of optimization methods in tackling computation indispensability. This approach made better use of memory, resulting in high-resolution shot experiments, with higher bit quantization compared to QLoRA on base model approach.

The above findings point out that shot number optimization together with the use of optimization techniques is key to high-quality text generation.

Prompt chaining with DSP and few-shots

The designed experiment conducts prompt chaining with few-shots and DSP techniques. The impact of diverse chain lengths and shot sequences on the model's capability to generate high-quality results is investigated.

Across the tested scenarios, a notable trend emerges: The increment of chain lengths along with a number of shots results in better results in measured metrics. Additionally, for the 1-3-shot situation with a chain length of 1 and 3 shots, the model has a BLEU score of 29.52. Moreover, extending the chain length while keeping the number of shots at the same level leads to an improvement in the results, which is illustrated in Table 7.5. An increase in chain length and number of shots by 1 further improved the results. This indicates that only increasing chain length improves the model performance, and adding an increase in shots adds to the improvements.

The results here imply that prompt stacking, along with few-shot and DSP methods, could lead to an improvement in the text generation's quality.

7.3 Agentic Workflow Results

In this experiment, the obtained results (refer to Table 7.6) did not yield the same level of success as observed in other experiments, notably the prompt engineering trials, owing to the constraints and limitations we encountered. Analysis of the results reveals that the one-shot prompt achieved the highest BLEU score, while increasing the number of shots did not notably improve model performance.

It's worth noting that the strength of the agentic workflow primarily lies in the tools provided to each agent. However, the implementation of these tools was hindered by the necessity to handle private data, thereby preventing their display in public domains such as web searches, which are among the most powerful tools available. Moreover, there remains ample opportunity for further exploration with this approach. The crew could potentially comprise

Table 7.3: Mistral 7B Instruct v0.2 in GGUF format with Q_5_K_M as quantization method using Llama-cpp-python

Shots	Matched	BLEU	Meteor	Rouge	Similarity
Half-shot	10	31.37	0.22	*R1*: 0.29 *R2*: 0.054 *RL*: 0.15 *RLsum*: 0.274	*Lev*: 0.22 *Jacc*: 0.71 *spaCy*: 0.78
1-shot	34	31.35	0.24	*R1*: 0.30 *R2*: 0.065 *RL*: 0.16 *RLsum*: 0.28	*Lev*: 0.22 *Jacc*: 0.71 *spaCy*: 0.79
2-shot	31	31.48	0.235	*R1*: 0.30 *R2*: 0.062 *RL*: 0.16 *RLsum*: 0.276	*Lev*: 0.227 *Jacc*: 0.716 *spaCy*: 0.791
3-shot	25	31.60	0.23	*R1*: 0.30 *R2*: 0.07 *RL*: 0.167 *RLsum*: 0.283	*Lev*: 0.23 *Jacc*: 0.72 *spaCy*: 0.79
4-shot	31	29.92	0.23	*R1*: 0.30 *R2*: 0.065 *RL*: 0.163 *RLsum*: 0.28	*Lev*: 0.22 *Jacc*: 0.71 *spaCy*: 0.79
5-shot	20	29.78	0.22	*R1*: 0.292 *R2*: 0.06 *RL*: 0.16 *RLsum*: 0.272	*Lev*: 0.22 *Jacc*: 0.71 *spaCy*: 0.77
3-shot with HPO	**26**	**32.33**	**0.23**	*R1*: **0.30** *R2*: **0.061** *RL*: **0.161** *RLsum*: **0.282**	*Lev*: **0.23** *Jacc*: **0.71** *spaCy*: **0.79**

diverse agents assigned tasks beyond those defined in this thesis, opening new avenues for experimentation and refinement.

7.4 Choosing a Final Model

The choice of the best model from our prompt engineering experiments was made after taking into account several aspects. Our aim was to find the best techniques and settings that showed the most promising results in the context of our study. The choice was based on the fact that prompt engineering always performed better than other methods that were being

Table 7.4: Mistral 7B Instruct v0.2 in GGUF format with Q_8_0 as quantization method using Llama-cpp-python

Shots	Matched	BLEU	Meteor	Rouge	Similarity
Half-shot	8	32.17	0.215	*R1*: 0.296 *R2*: 0.0055 *RL*: 0.155 *RLsum*: 0.276	*Lev*: 0.22 *Jacc*: 0.71 *spaCy*: 0.78
1-shot	35	30.44	0.24	*R1*: 0.29 *R2*: 0.067 *RL*: 0.163 *RLsum*: 0.28	*Lev*: 0.23 *Jacc*: 0.71 *spaCy*: 0.79
2-shot	36	31.59	0.232	*R1*: 0.297 *R2*: 0.061 *RL*: 0.159 *RLsum*: 0.279	*Lev*: 0.232 *Jacc*: 0.715 *spaCy*: 0.791
3-shot	24	30.66	0.227	*R1*: 0.293 *R2*: 0.06 *RL*: 0.161 *RLsum*: 0.274	*Lev*: 0.22 *Jacc*: 0.71 *spaCy*: 0.78
4-shot	23	30.51	0.23	*R1*: 0.30 *R2*: 0.065 *RL*: 0.164 *RLsum*: 0.279	*Lev*: 0.22 *Jacc*: 0.72 *spaCy*: 0.788
5-shot	30	28.74	0.22	*R1*: 0.29 *R2*: 0.06 *RL*: 0.16 *RLsum*: 0.272	*Lev*: 0.22 *Jacc*: 0.71 *spaCy*: 0.78
Half-shot with HPO	**9**	**32.925**	**0.212**	*R1*: **0.292** *R2*: **0.053** *RL*: **0.152** *RLsum*: **0.272**	*Lev*: **0.226** *Jacc*: **0.71** *spaCy*: **0.775**

considered.

In order to continue with the human evaluation, we selected from the three following approaches that showed the most promising results in model performance:

- DSP with Few-shot Prompting on the Base Model with QLoRA
- DSP with Few-shot Prompting on GGUF Model through llama-cpp-python
- DSP with Few-shot and Chain Prompting on GGUF Model through llama-cpp-python

We were trying to diversify the results and compare the performance of these models by using standard metrics such as BLEU and ROUGE against the real-life evaluations of the hu-

Table 7.5: Prompt chaining using Mistral 7B Instruct v0.2 in GGUF format with Q_5_K_M

Chain length & Number of shots	Matched	BLEU	Meteor	Rouge	Similarity
1 & 3-shots	27	29.52	0.208	*R1*: 0.28 *R2*: 0.05 *RL*: 0.15 *RLsum*: 0.26	*Lev*: 0.22 *Jacc*: 0.71 *spaCy*: 0.78
2 & 3-shots	26	30	0.21	*R1*: 0.28 *R2*: 0.05 *RL*: 0.15 *RLsum*: 0.26	*Lev*: 0.21 *Jacc*: 0.71 *spaCy*: 0.78
3 & 4-shots	**28**	**31.15**	**0.21**	*R1*: **28** *R2*: **0.05** *RL*: **0.15** *RLsum*: **0.26**	*Lev*: **0.21** *Jacc*: **0.71** *spaCy*: **0.78**

Table 7.6: Agentic workflow few-shot result table on Mistral 7B Instruct v0.2 pulled from Ollama

Shots	Matched	BLEU	Meteor	Rouge	Similarity
1-shot	**24**	**24.05**	**0.178**	*R1*: **0.206** *R2*: **0.025** *RL*: **0.18** *RLsum*: **0.114**	*Lev*: **0.203** *Jacc*: **0.7** *spaCy*: **0.74**
2-shot	14	23.04	0.171	*R1*: 0.195 *R2*: 0.07 *RL*: 0.105 *RLsum*: 0.184	*Lev*: 0.20 *Jacc*: 0.69 *spaCy*: 0.75
3-shot	13	22.7	0.175	*R1*: 0.197 *R2*: 0.024 *RL*: 0.109 *RLsum*: 0.186	*Lev*: 0.19 *Jacc*: 0.69 *spaCy*: 0.76
4-shot	12	22.2	0.178	*R1*: 0.197 *R2*: 0.026 *RL*: 0.109 *RLsum*: 0.186	*Lev*: 0.19 *Jacc*: 0.7 *spaCy*: 0.75
5-shot	19	23.2	0.166	*R1*: 0.197 *R2*: 0.023 *RL*: 0.108 *RLsum*: 0.188	*Lev*: 0.197 *Jacc*: 0.7 *spaCy*: 0.74

man testers. This method helped us to find out whether higher metric scores were associated with better model performance or if some other factors should be taken into account, with metrics acting as guides.

After the analysis, the results showed that there is a strong connection between the higher scores in standard metrics and the increased human evaluation scores. The model with the best metric scores always won in the human assessment tests, thus proving the usefulness of these metrics as the predictors of the model's performance.

In our human evaluation process, we included real test cases randomly in the sample pool to establish the reference value for human evaluation metrics based on the quality of the existing test cases. Our highest human evaluation score was about 20 percent less than the reference value, see Table 7.7. Even though this variation occurred, these test cases were still usable as the reference value was based on real test cases that had been reviewed and validated before, so the quality of the baseline was ensured.

Through this entire process of model selection and evaluation, we were able to obtain valuable insights into the performance of various prompt engineering techniques and their real-world applicability.

Table 7.7: Human evaluation result table on the top three optimization approaches

Human Evaluation Scores *Reference score for real test cases: 4.68*	
Approach	**Human Evaluation Score/BLEU Score**
DSP with Few-shot Prompting on the Base Model with QLoRA	3.44/30.18
DSP with Few-shot Prompting on GGUF Model through llama-cpp-python	**3.71/32.93**
DSP with Few-shot and Chain Prompting on GGUF Model through llama-cpp-python	3.45/31.15

To sum up, our detailed evaluation process has been the foundation of the discovery of the most suitable model for the generation of contextually relevant and usable test case documents based on the given specification documents. After carefully examining different techniques and configurations, it turned out that the DSP model with half-shot prompting on the 8-bit (Q8) GGUF Model through llama-cpp-python always beat the others. This method not only got the best scores in the standard metrics such as BLEU and ROUGE but also did well in the human evaluation metrics. We are sure that it can fulfill the difficult conditions of our task, thus, showing the effectiveness of our quick engineering method in improving the model performance for real-life applications.

7.5 Building a Practical Tool

We developed a tool that utilizes our optimized model to automate the process of test case generation from feature documents. The tool takes two arguments, firstly a path to a txt file that contains the feature document text and the second argument is a value that decides if the tool should use GPU or not. The tool extracts the text from the txt file, applies our optimized prompt template, and subsequently feeds the prompt to our highest-scoring model. The response from the model is then saved in an output folder as a txt file containing the generated test cases. The test cases generated in the text file can serve as a draft version. Test engineers can refine them to create a final document of test cases, which can then be integrated for use as functional test cases.

Chapter 8
Conclusion

Our thesis focuses on the use of NLP techniques in the software testing domain, providing valuable insights into their performance and downsides. We evaluate three main approaches: fine-tuning, prompt engineering, and AI agentic workflow, conducting over 615 experiment runs with various settings. Additionally, we implemented a primary pipeline for LLM evaluation and hyperparameter optimization (see Appendix A). '

Our approach achieves close to state-of-the-art performance, with a BLEU score of 32.93 compared to the BLEU scores of 28.4 for English-to-German and 41.0 for English-to-French translation tasks (Vaswani et al., 2017). In human evaluations, our approach scored 3.71, which is close to the human-written reference score of 4.68, pointing to its potential effectiveness despite not undergoing planning and review phases.

Reflections on research expectations. For completeness, we reiterate our research expectations and provide the corresponding answers and reflections.

- Can a small-sized large language model (7 billion parameters) with quantization be effectively applied in the software domain using consumer-grade GPU hardware?
  - Our research demonstrates that a small-sized large language model (7 billion parameters) with quantization can be effectively utilized with consumer-grade GPU hardware. We evaluated two different quantization methods, QLoRA and GGUF format. Through these methods, we successfully applied Mistral 7B on the available single GPU we had for this thesis, which is RTX 4080 with 16 GB VRAM.
- Is it feasible to generate test cases based on feature descriptions using a small-sized large language model?
  - The comparison of results suggests that further work on NLP-generated test cases is needed. Despite the superiority of machine-generated test cases in certain aspects, there are also cases where traditional human-written test cases remain advantageous. Particularly in relation to the specific details found in the planning and review phases. Nevertheless, the potential for leveraging NLP technologies for

automating the testing process within the area of software testing is very high – this could be a promising direction for future developments.

- How does this approach compare in effectiveness to human-written test cases that have undergone planning and reviewing phases?
  - This thesis demonstrates that machine-generated test cases can approach the effectiveness of human-written ones. During human evaluation, our approach scored 3.71, compared to 4.68 for human-written test cases. Additionally, our method achieved a BLEU score of 32.93, surpassing the state-of-the-art machine translation from English-to-German, which scored 28.4, and coming close to the state-of-the-art English-to-French translation, which scored 41.0.

Future work. Future research in this area might put in focus a number of areas that would be helpful for further improvement of the quality of NLP-generated test cases. With that, research into higher and more complex LLMs can prove potentially useful, for example by focusing on further enhancing test case generation. Secondly, replacing older mathematical approaches with state-of-the-art quantization algorithms and optimization algorithms could help boost performance and speed on commodity-grade hardware. Further research might be useful to investigate why fine-tuning was not suitable in this case and the possible improvements and modifications that can be made to make this approach more effective.

In addition, the AI agentic workflow could be explored by incorporating variations of crew structure and even perhaps the use of crew tools that contribute to coordination between the agents and increased efficiency. Additionally, introducing human-in-the-loop interaction with the AI agents would improve the performance to a large extent. There is an opportunity to use a much bigger real dataset to increase overall performance, as our thesis was based only on 93 real data samples, while the rest was augmented.

Furthermore, a solid future strategy could also include the use of RAG to be able to consider and read the specifications of some of the features in the other types of formats besides text (i.e., web pages, pictures, and diagrams) and therefore create test cases in appropriate formats accordingly.

References

Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. (2023). GQA: Training generalized multi-query transformer models from multi-head checkpoints.

Al-Hossami, E. and Shaikh, S. (2022). A survey on artificial intelligence for source code: a dialogue systems perspective. *arXiv preprint arXiv:2202.04847*.

Alaqail, H. and Ahmed, S. (2018). Overview of software testing standard iso/iec/ieee 29119. *International Journal of Computer Science and Network Security (IJCSNS)*, 18(2):112–116.

Alvarez, J. E. (2017). A review of word embedding and document similarity algorithms applied to academic text. Master's thesis, University of Freiburg.

Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., McMinn, P., Bertolino, A., et al. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of systems and software*, 86(8):1978–2001.

Ansari, A., Shagufta, M. B., Sadaf Fatima, A., and Tehreem, S. (2017). Constructing test cases using natural language processing. In *2017 Third International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB)*, pages 95–99.

Anthropic (2024). Prompt chaining prompt engineering guide. `https://docs.anthropic.com/claude/docs/chain-prompts`.

Aoyama, Y., Kuroiwa, T., and Kushiro, N. (2021). Executable test case generation from specifications written in natural language and test execution environment. In *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6.

Arvidsson, S. and Axell, J. (2023). Prompt engineering guidelines for LLMs in requirements engineering. Master's thesis, Gothenburg University.

Ayenew, H. and Wagaw, M. (2024). Software test case generation using natural language processing (nlp): A systematic literature review. *Artificial Intelligence Evolution*, pages 1–10.

Banjara, B. (2024). A comprehensive guide to fine-tuning large language models. URL: https://www.analyticsvidhya.com/blog/2023/08/fine-tuning-large-language-models/.

Bao, H., Dong, L., Wei, F., Wang, W., Yang, N., Liu, X., Wang, Y., Piao, S., Gao, J., Zhou, M., and Hon, H.-W. (2020). Unilmv2: pseudo-masked language models for unified language model pre-training. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org.

Bengio, Y., Ducharme, R., and Vincent, P. (2000). A neural probabilistic language model. In Leen, T., Dietterich, T., and Tresp, V., editors, *Advances in Neural Information Processing Systems*, volume 13. MIT Press.

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. (2011a). Algorithms for hyper-parameter optimization. In Shawe-Taylor, J., Zemel, R., Bartlett, P., Pereira, F., and Weinberger, K., editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc.

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. (2011b). Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24. URL: https://www.analyticsvidhya.com/blog/2023/08/fine-tuning-large-language-models/.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Catterall, W. A., Perez-Reyes, E., Snutch, T. P., and Striessnig, J. (2005). International union of pharmacology. xlviii. nomenclature and structure-function relationships of voltage-gated calcium channels. *Pharmacological reviews*, 57(4):411–425.

Celikyilmaz, A., Clark, E., and Gao, J. (2021). Evaluation of text generation: A survey. URL: https://arxiv.org/abs/2006.14799.

Chauhan, N. S. (2022). Openai gpt-3: Understanding the architecture. URL: https://www.spaceo.ai/blog/openai-gpt-architecture/.

Chen, H., Han, W., Yang, D., and Poria, S. (2022). Doublemix: Simple interpolation-based data augmentation for text classification. URL: https://arxiv.org/abs/2209.05297.

Choi, J., Jin, K., Lee, J., Song, S., and Kim, Y. (2024). Softeda: Rethinking rule-based data augmentation with soft labels. URL: https://arxiv.org/abs/2402.05591.

Cremers, A. and Ginsburg, S. (1975). Context-free grammar forms. *Journal of Computer and System Sciences*, 11(1):86–117. URL: https://www.sciencedirect.com/science/article/pii/S0022000075800511.

de Kock, N., Rautenbach, V., and Fabris-Rotelli, I. (2022). Towards an open source python library for automated exploratory spatial data analysis. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 43:91–98.

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. (2023). Qlora: Efficient finetuning of quantized llms. URL: `https://arxiv.org/abs/2305.14314`.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. URL: `https://arxiv.org/abs/1810.04805`.

Dong, L., Yang, N., Wang, W., Wei, F., Liu, X., Wang, Y., Gao, J., Zhou, M., and Hon, H.-W. (2019). Unified language model pre-training for natural language understanding and generation.

Du, Z., Qian, Y., Liu, X., Ding, M., Qiu, J., Yang, Z., and Tang, J. (2022). Glm: General language model pretraining with autoregressive blank infilling.

El-Kassas, W., Salama, C., Rafea, A., and Mohamed, H. (2020). Automatic text summarization: A comprehensive survey. *Expert Systems with Applications*, 165:113679.

Elfwing, S., Uchibe, E., and Doya, K. (2018). Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural Networks*, 107:3–11. Special issue on deep reinforcement learning.

Feng, S. Y., Gangal, V., Wei, J., Chandar, S., Vosoughi, S., Mitamura, T., and Hovy, E. (2021). A survey of data augmentation approaches for nlp.

Ferrario, A. and Nägelin, M. (2020). The art of natural language processing: classical, modern and contemporary approaches to text document classification. *Modern and Contemporary Approaches to Text Document Classification (March 1, 2020)*.

Friederich, S. (2017). Fine-tuning. *The Stanford encyclopedia of philosophy*. URL: `https://plato.stanford.edu/entries/fine-tuning/.`

Gangal, V., Feng, S. Y., Alikhani, M., Mitamura, T., and Hovy, E. (2022). Nareor: The narrative reordering problem. URL: `https://arxiv.org/abs/2104.06669`.

Gillioz, A., Casas, J., Mugellini, E., and Khaled, O. A. (2020). Overview of the transformer-based models for nlp tasks. In *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*, pages 179–183.

Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J., and Sculley, D. (2017). Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1487–1495.

Guo, T., Chen, X., Wang, Y., Chang, R., Pei, S., Chawla, N. V., Wiest, O., and Zhang, X. (2024). Large language model based multi-agents: A survey of progress and challenges.

Gupta, A. (2023). Test case selection from test specifications using natural language processing. Master's thesis, Stockholm University.

Gupta, N., Sharma, A., and Pachariya, M. K. (2019). An insight into test case optimization: Ideas and trends with future perspectives. *IEEE Access*, 7:22310–22327.

Haaralahti, E. (2024). Utilization of local large language models for business applications. URL: https://aaltodoc.aalto.fi/server/api/core/bitstreams/1b63c648-44b3-46a9-9385-059d2584abea/content.

Harris, D., Martin, G. M., Perera, I., and Poskitt, D. S. (2017). Construction and visualization of optimal confidence sets for frequentist distributional forecasts.

Hassan, A. E., Lin, D., Rajbahadur, G. K., Gallaba, K., Cogo, F. R., Chen, B., Zhang, H., Thangarajah, K., Oliva, G. A., Lin, J., et al. (2024). Rethinking software engineering in the era of foundation models: A curated catalogue of challenges in the development of trustworthy fmware. *arXiv preprint arXiv:2402.15943*.

Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. (2020). The curious case of neural text degeneration. URL: https://arxiv.org/abs/1904.09751.

Hong, C.-H., Spence, I., and Nikolopoulos, D. S. (2017). Gpu virtualization and scheduling methods: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 50(3):1–37.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. (2021). Lora: Low-rank adaptation of large language models.

ISO/IEC/IEEE International Standard, S. t. (2013). Iso/iec/ieee international standard - software and systems engineering —software testing —part 1:concepts and definitions. *ISO/IEC/IEEE 29119-1:2013(E)*, pages 1–64.

Jain, N., yeh Chiang, P., Wen, Y., Kirchenbauer, J., Chu, H.-M., Somepalli, G., Bartoldson, B. R., Kailkhura, B., Schwarzschild, A., Saha, A., Goldblum, M., Geiping, J., and Goldstein, T. (2023). Neftune: Noisy embeddings improve instruction finetuning.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. (2023a). Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. (2023b). Mistral 7b.

Katz, S. (1987). Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(3):400–401.

Khawaja, R. (2023). Fine-tuning llms 101. URL: https://datasciencedojo.com/blog/fine-tuning-llms/.

Kiran, A., Butt, W. H., Anwar, M. W., Azam, F., and Maqbool, B. (2019). A comprehensive investigation of modern test suite optimization trends, tools and techniques. *IEEE Access*, 7:89093–89117.

Lafi, M., Alrawashed, T., and Hammad, A. M. (2021). Automated test cases generation from requirements specification. In *2021 International Conference on Information Technology (ICIT)*, pages 852–857.

Lambert, E., Fiers, M., Nizamov, S., Tassaert, M., Johnson, S. G., Bienstman, P., and Bogaerts, W. (2010). Python bindings for the open source electromagnetic simulator meep. *Computing in Science & Engineering*, 13(3):53–65.

Laskin, M., Wang, L., Oh, J., Parisotto, E., Spencer, S., Steigerwald, R., Strouse, D., Hansen, S., Filos, A., Brooks, E., et al. (2022). In-context reinforcement learning with algorithm distillation. *arXiv preprint arXiv:2210.14215*.

Lavie, A., Sagae, K., and Jayaraman, S. (2004). The significance of recall in automatic metrics for mt evaluation. In *Machine Translation: From Real Users to Research: 6th Conference of the Association for Machine Translation in the Americas, AMTA 2004, Washington, DC, USA, September 28-October 2, 2004. Proceedings 6*, pages 134–143. Springer.

Leviathan, Y., Kalman, M., and Matias, Y. (2023). Fast inference from transformers via speculative decoding. URL: `https://arxiv.org/abs/2211.17192`.

Leygonie, R., Lobry, S., Vimont, G., and Wendling, L. (2023). Transforming multidimensional data into images to overcome the curse of dimensionality. In *2023 IEEE International Conference on Image Processing (ICIP)*, pages 700–704.

Li, J., Tang, T., Zhao, W. X., Nie, J.-Y., and Wen, J.-R. (2022). Pretrained language models for text generation: A survey.

Li, Z., Peng, B., He, P., Galley, M., Gao, J., and Yan, X. (2023). Guiding large language models via directional stimulus prompting.

Lin, C.-Y. (2004). Rouge: A package for automatic evaluation of summaries. URL: `https://aclanthology.org/W04-1013/`.

Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., and Han, S. (2023). Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*.

Lunney, D., Vieira, N., Audi, G., Gaulard, C., de Saint Simon, M., and Thibault, C. (2006). Mass measurements of the shortest-lived nuclides a la mistral. *International Journal of Mass Spectrometry*, 251(2-3):286–292.

Maclaren, U. (2024). Do you know when to use 0-shot, 1-shot, or multi-shot prompts (e.g. give it 1 or more examples)? URL: `https://anilktalla.medium.com/prompt-engineering-1-shot-prompting-283a0b2b1467#:~:text=1.,example%20can%20guide%20the%20output.`

Microsoft (2021). Neural Network Intelligence. URL: `https://nni.readthedocs.io/en/stable/`.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. In Bengio, Y. and LeCun, Y., editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*.

Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., and Gao, J. (2024). Large language models: A survey. *arXiv preprint arXiv:2402.06196*.

Mittal, S. and Vetter, J. S. (2014). A survey of methods for analyzing and improving gpu energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2):1–23. URL: `https://arxiv.org/abs/1904.09751`.

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

Niu, Z., Zhong, G., and Yu, H. (2021). A review on the attention mechanism of deep learning. *Neurocomputing*, 452:48–62.

Otero-Escobar, A. D. and Velasco-Ramírez, M. L. (2023). Study on exploratory data analysis applied to education. In *2023 IEEE International Conference on Engineering Veracruz (ICEV)*, pages 1–5.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In Isabelle, P., Charniak, E., and Lin, D., editors, *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Patil, R., Boit, S., Gudivada, V., and Nandigam, J. (2023). A survey of text representation and embedding techniques in nlp. *IEEE Access*, 11:36120–36146.

Pennington, J., Socher, R., and Manning, C. (2014). GloVe: Global vectors for word representation. In Moschitti, A., Pang, B., and Daelemans, W., editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.

Pham, Q., Do, G., Nguyen, H., Nguyen, T., Liu, C., Sartipi, M., Nguyen, B. T., Ramasamy, S., Li, X., Hoi, S., and Ho, N. (2024). Competesmoe – effective training of sparse mixture of experts via competition.

Potuzak, T. and Lipka, R. (2023). Current trends in automated test case generation. In *2023 18th Conference on Computer Science and Intelligence Systems (FedCSIS)*, pages 627–636. IEEE.

Qiu, X., Sun, T., Xu, Y., Shao, Y., Dai, N., and Huang, X. (2020). Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, 63(10):1872–1897.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. (2018). Improving language understanding by generative pre-training. *OpenAI blog*.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2023). Exploring the limits of transfer learning with a unified text-to-text transformer.

Ramlochan, S. (2023). Getting started with prompt chaining. URL: `https://promptengineering.org/getting-started-with-prompt-chaining/`.

Reiter, E. and Dale, R. (1997). Building applied natural language generation systems. *Natural Language Engineering*, 3(1):57–87.

Reitz, K. (2023). Requests: Http for humans™. URL: `https://requests.readthedocs.io/en/latest/`.

Renze, M. and Guven, E. (2024). The effect of sampling temperature on problem solving in large language models. *arXiv preprint arXiv:2402.05201*.

Richardson, L. (2024). Beautiful soup documentation. URL: `https://www.crummy.com/software/BeautifulSoup/bs4/doc/`.

Romanus Myrberg, N. and Danielsson, S. (2023). Question-answering in the financial domain. Master's thesis, Lund University.

Rothe, S., Narayan, S., and Severyn, A. (2020). Leveraging pre-trained checkpoints for sequence generation tasks. *Transactions of the Association for Computational Linguistics*, 8:264–280.

Salman, A. (2020). Test case generation from specifications using natural language processing. Master's thesis, KTH Royal Institute of Technology.

Saxena, A. (2023). Prompt lookup decoding. URL: `https://github.com/apoorvumang/prompt-lookup-decoding/`.

Sennrich, R., Haddow, B., and Birch, A. (2016). Improving neural machine translation models with monolingual data.

Shazeer, N. (2019). Fast transformer decoding: One write-head is all you need. URL: `https://arxiv.org/abs/1911.02150`.

Singh, S. K. and Singh, A. (2012). *Software testing*. Vandana Publications. URL: `https://fall14se.wordpress.com/wp-content/uploads/2017/12/software-testing-yogesh-singh.pdf`.

Sinha, S., Zhao, Z., Goyal, A., Raffel, C., and Odena, A. (2020). Top-k training of gans: Improving gan performance by throwing away bad samples.

Sivarajkumar, S., Kelley, M., Samolyk-Mazzanti, A., Visweswaran, S., and Wang, Y. (2024). An empirical evaluation of prompting strategies for large language models in zero-shot clinical natural language processing: Algorithm development and validation study. *JMIR Medical Informatics*, 12:e55318.

Song, Y., Schwing, A. G., Zemel, R. S., and Urtasun, R. (2016). Training deep neural networks via direct loss minimization.

Stenberg, D. (2024). Documentation overview. URL: `https://danielsieger.com/blog/2023/04/24/framework-for-better-documentation.html`.

Stern, M., Shazeer, N., and Uszkoreit, J. (2018). Blockwise parallel decoding for deep autoregressive models.

Stiny, G. (1992). Weights. *Environment and planning B: Planning and design*, 19(4):413–430. URL: https://www.andrew.cmu.edu/course/48747/subFrames/readings/Stiny.weights.pdf.

Takyar, A. (2023). Optimizing pre-trained models: A guide to parameter-efficient fine-tuning (peft). URL: https://www.leewayhertz.com/parameter-efficient-fine-tuning/.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. (2023). Llama: Open and efficient foundation language models.

Tribes, C., Benarroch-Lelong, S., Lu, P., and Kobyzev, I. (2024). Hyperparameter optimization for large language model instruction-tuning.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Wachter, S. and Mittelstadt, B. (2019). A right to reasonable inferences: re-thinking data protection law in the age of big data and ai. *Colum. Bus. L. Rev.*, page 494.

Wang, C., Pastore, F., Goknil, A., and Briand, L. C. (2020). Automatic generation of acceptance test cases from use case specifications: an nlp-based approach. *IEEE Transactions on Software Engineering*, 48(2):585–616.

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., and Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*.

Waskom, M. L. (2021). seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021.

Wes McKinney (2010). Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61.

Whittaker, J. A. (2000). What is software testing? and why is it so hard? *IEEE software*, 17(1):70–79.

Wolfe, C. R. (2023). Supervised fine-tuning (SFT) for language models. https://cameronrwolfe.substack.com/p/understanding-and-using-supervised.

Xu, L., Xie, H., Qin, S.-Z. J., Tao, X., and Wang, F. L. (2023). Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment. URL: https://arxiv.org/pdf/2312.12148.

Yang, G., Haque, M., Song, Q., Yang, W., and Liu, X. (2022). Testaug: A framework for augmenting capability-based nlp tests.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R., and Le, Q. V. (2020). Xlnet: Generalized autoregressive pretraining for language understanding.

Zaib, M., Sheng, Q. Z., and Zhang, W. E. (2021). A short survey of pre-trained language models for conversational ai-a newage in nlp.

Zhang, S., Dong, L., Li, X., Zhang, S., Sun, X., Wang, S., Li, J., Hu, R., Zhang, T., Wu, F., and Wang, G. (2024). Instruction tuning for large language models: A survey.

Zhelezniak, V., Savkov, A., Shen, A., and Hammerla, N. Y. (2019). Correlation coefficients and semantic textual similarity. *arXiv preprint arXiv:1905.07790*.

Appendices

Appendix A

LLM Evaluation and Hyperparameter Optimization Pipeline

As part of this thesis, we built a pipeline (see Figure A.1) to automate multiple functions, such as generating responses, evaluating them, utilizing the NNI framework for hyperparameter optimization, and finally choosing the best model with the highest BLEU score.

The pipeline is broken down as follows:

LLM and Test Data Loading The pipeline starts by loading a chosen LLM and a test dataset. We built the LLM based on the chosen draft model, the quantized GGUF model, and the chat format template, using llama-cpp-python. Then we loaded both the test dataset and the example-shots dataset for prompt engineering.

Response Generation The LLM then generates, with a given prompt, responses for each sample within the test dataset. Additionally, we defined a search space for the hyperparameters, a tuner, and an assessor for the HPO phase later.

Evaluation The generated responses are evaluated using predefined metrics. Then, the BLEU score of each sample evaluation is reported to the NNI as an intermediate result.

Intermediate BLEU Score Reporting The median BLEU score is used as the default metric for the NNI, where the objective of the tuner is to maximize this metric. Moreover, the assessor here works as the early stopping mechanism.

Average BLEU Score Calculation Once all the samples in the test dataset are evaluated, the pipeline calculates the average BLEU score for the entire trial. This score is then reported to NNI as the final result.

Best Model Selection Finally, the pipeline picks the model that achieved the highest BLEU score during the optimization process as the best model.

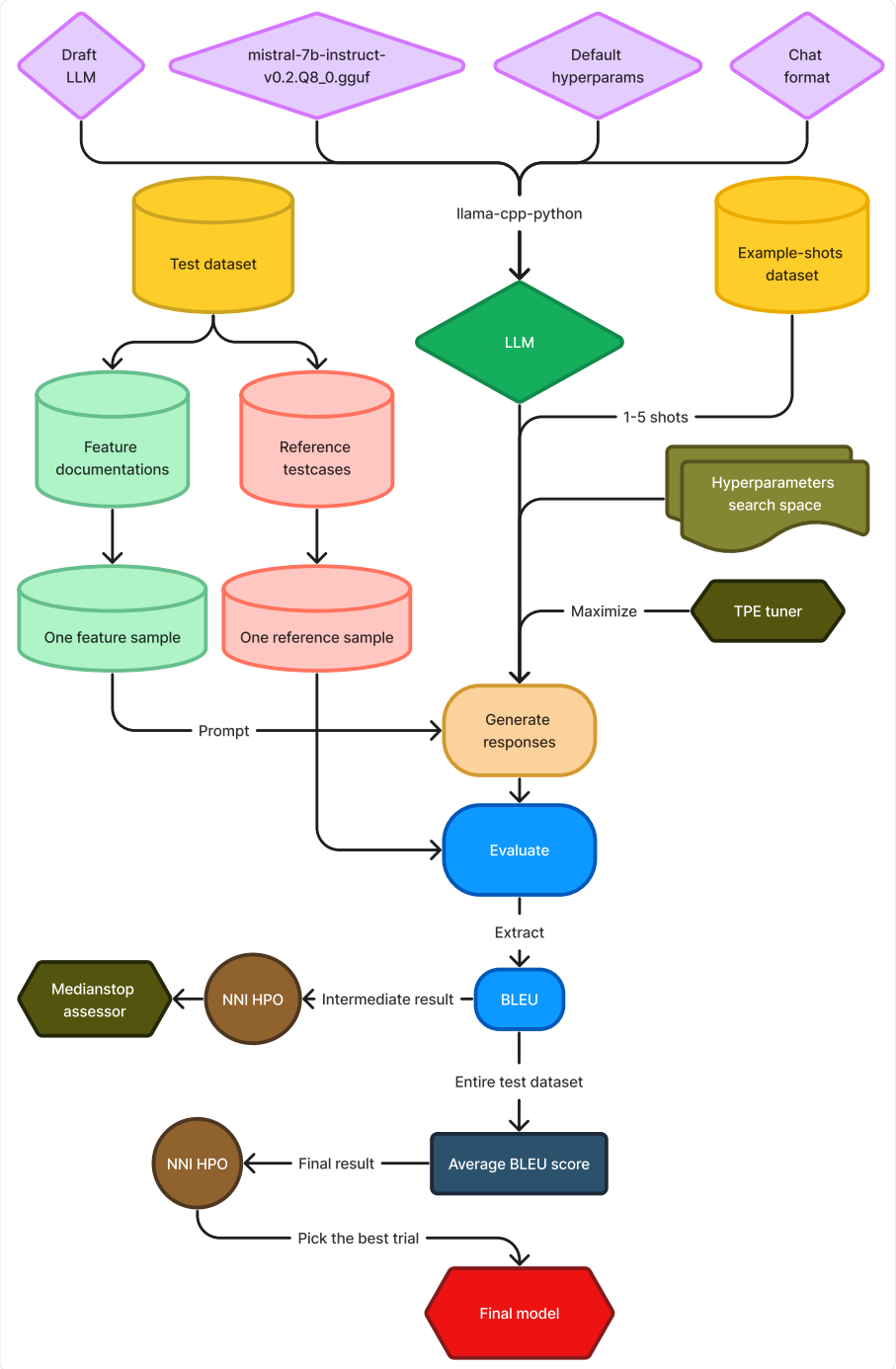

Figure A.1: The LLM evaluation and HPO pipeline created during this thesis.

Appendix B
Experiments Tracking

In our HPO experiments, specifically for the top two promising models, we meticulously document various aspects, through NNI, to ensure comprehensive analysis and reproducibility. The first step involves defining the search space, which includes specifying the range and type of hyperparameters to be optimized. For each experiment, we record the specific hyperparameter settings tested, the corresponding model performance metrics, and any notable observations.

We also generate and analyze graphs to visualize the results (see Figures B.6, & B.5), such as performance trends over different hyperparameter values, convergence rates, and comparison plots of different optimization techniques. These graphs help in identifying the most effective hyperparameter configurations and understanding their impact on model performance. Additionally, we document the computational resources used, the duration of each experiment, and any encountered issues or anomalies. This detailed tracking approach ensures that the experiments are comprehensive, and the insights gained are valuable for improving the outcome of this thesis.

Listing B.1: Defining the search space for lama-cpp-python approach.

```
temp_list = np.arange(0,1.5 + 0.1, 0.1).tolist()
search_space_llama_cpp = {
    'temperature': {'_type' : 'choice', '_value' :temp_list},
    'max_tokens': {'_type' : 'choice', '_value' :[1024, 2048, 4096,
       8192, 16384]},
    'num_pred_tokens': {'_type' : 'choice', '_value' :[5, 10, 15, 20,
       25, 30]},
    'top_p': {'_type' : 'uniform', '_value' :[0, 1]},
    'top_k': {'_type' : 'choice', '_value' :[1, 5, 10, 15, 20, 25, 30,
       35, 40]}

}
```

Mistral-7B-Instruct 5-bit-GGUF HPO with 3-shot In this experiment, we ran our pipeline with 80 trials and a predefined search space. This experiment took around two days (see Figure B.1) and the best trial had a 32.33 BLEU score with a specific combination of hyperparameter values, as Table B.1 demonstrates. Even though each trial had a BLEU score, we looked at its intermediate results and graphs to see how it did compared to other trials (see Figure B.2).

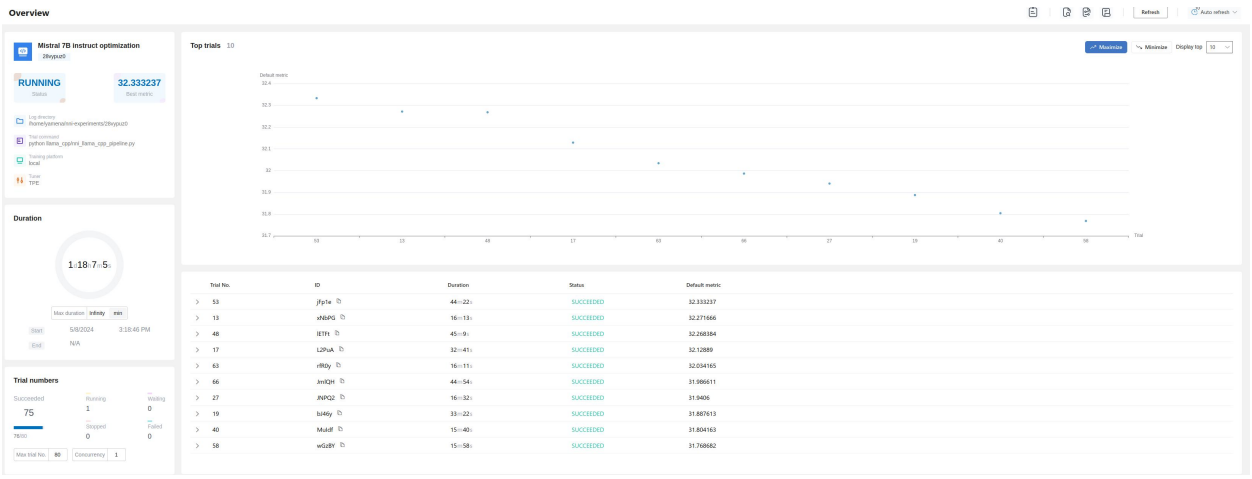

Figure B.1: Overview of Mistral-7B-Instruct 5-bit-GGUF HPO with 3-shot experiment.

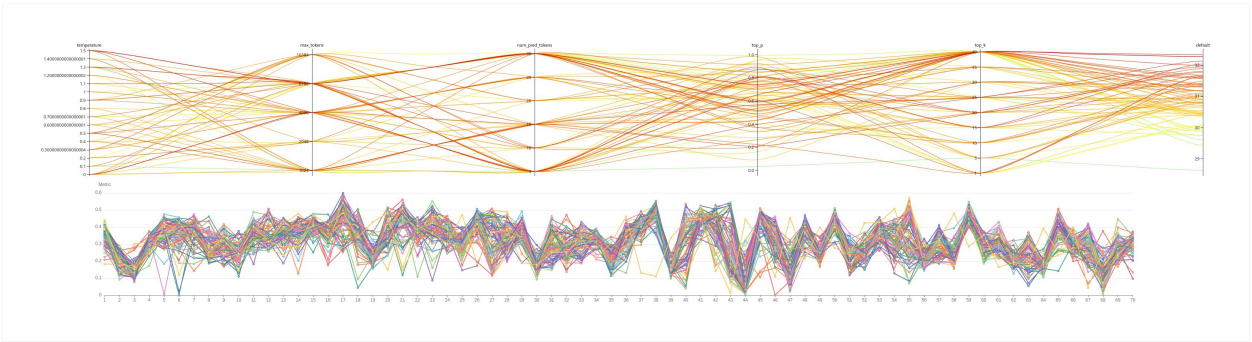

Figure B.2: Hyperparameters graph together with intermediate results for all trials. Default refers to the BLEU score.

Table B.1: Hyperparameters values of best trail for Mistral-7B-Instruct 5-bit-GGUF HPO with 3-shot.

Hyperparameter	Chosen Value
Temperature	1.5
Max Tokens	8192
Number of Predicted Tokens	30
top_p	0.74312
top_k	40

Mistral-7B-Instruct 8-bit-GGUF HPO with half-shot In this experiment, we ran our pipeline with 50 trials and a predefined search space. This experiment took around one day (see Figure B.3) and the best trial had a 32.925 BLEU score with a specific combination of hyperparameter values, as Table B.2 demonstrates. Even though each trial had a BLEU score, we looked at its intermediate results and graphs to see how it did compared to other trials (see Figure B.4).

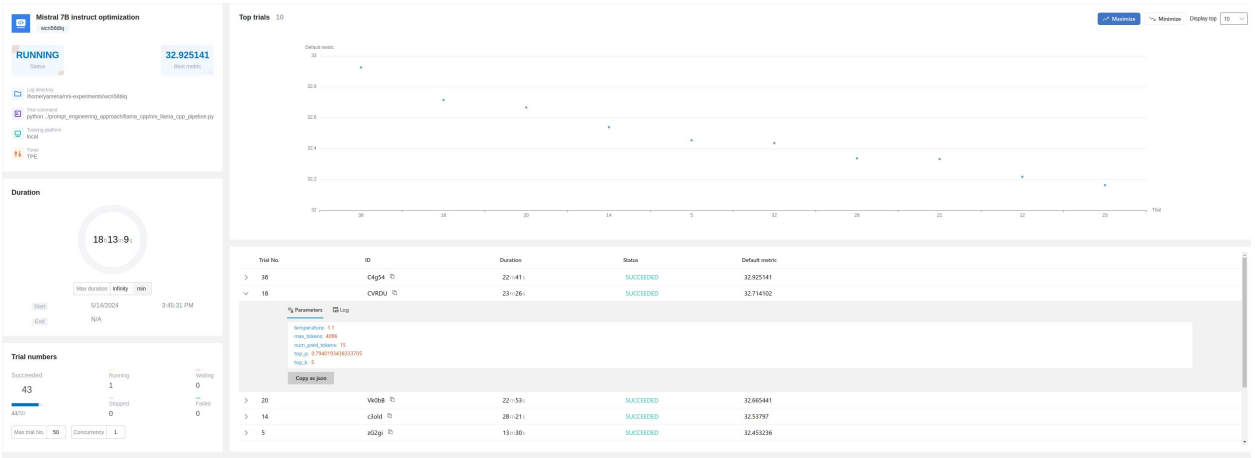

Figure B.3: Overview of Mistral-7B-Instruct 8-bit-GGUF HPO with half-shot experiment.

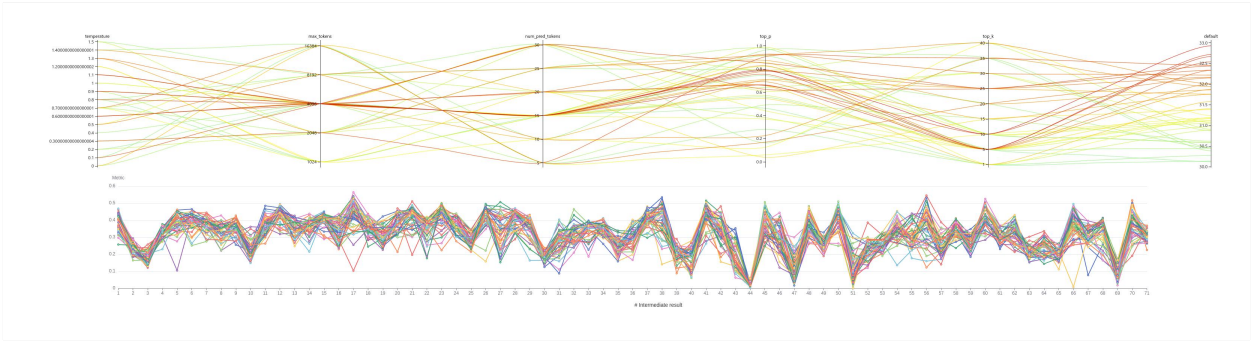

Figure B.4: Hyperparameters graph together with intermediate results for all trials. Default refers to the BLEU score.

Table B.2: Hyperparameters values of best trail for Mistral-7B-Instruct 8-bit-GGUF HPO with half-shot.

Hyperparameter	Chosen Value
Temperature	0.9
Max Tokens	4096
Number of Predicted Tokens	15
top_p	0.7859
top_k	10

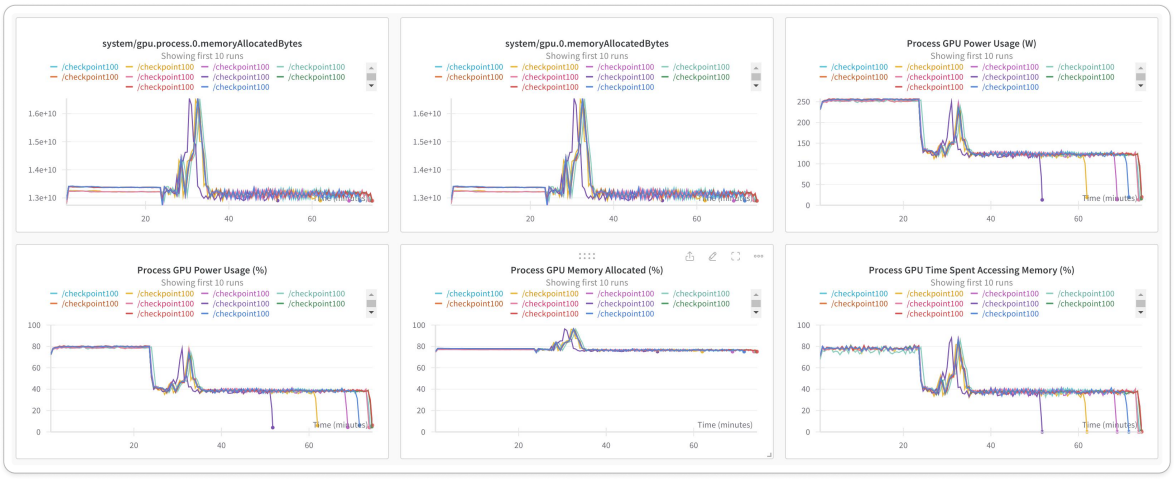

Figure B.5: Computational resources tracking over 615 runs using Wandb.

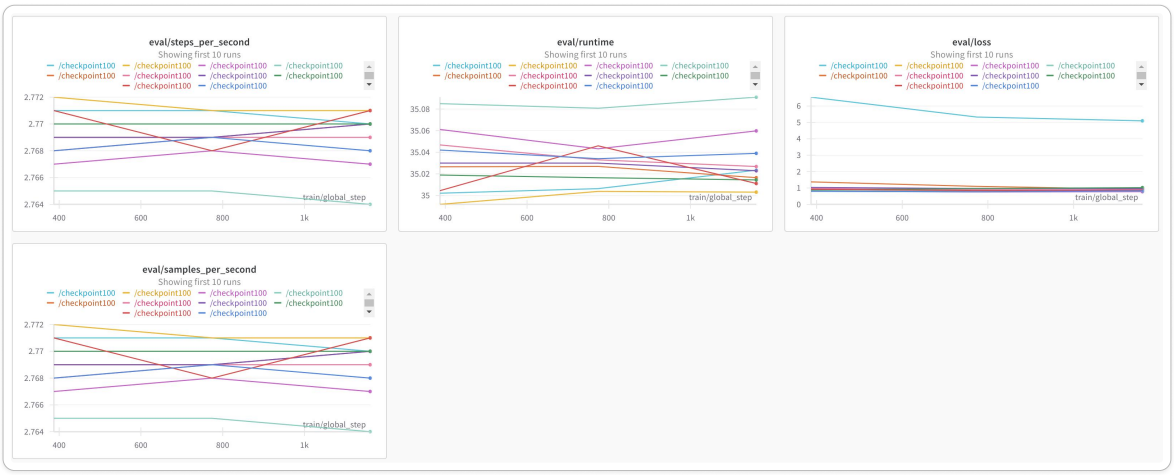

Figure B.6: Evaluation metrics tracking over 615 runs using Wandb.