

Master’s Programme in Security and Cloud Computing

# Fine-Tuning Large Language Models for Automating NEF API Calls

---

**Zainab Khan**

© 2025

This work is licensed under a [Creative Commons](https://creativecommons.org/licenses/by-nc-sa/4.0/) “Attribution-NonCommercial-ShareAlike 4.0 International” license.



---

**Author** Zainab Khan

---

**Title** Fine-Tuning Large Language Models for Automating NEF API Calls

---

**Degree programme** Master’s Programme in Security and Cloud Computing

---

**Major** Security and Cloud Computing

---

**Supervisors** Arto Hellas (Aalto University) , Ahmed Hussain (KTH University)

---

**Advisors** Mukesh Thakur (Ericsson), Panagiotis Papadimitratos (KTH University)

---

**Collaborative partner** Oy LM Ericsson

---

**Date** 23 March 2025

**Number of pages** 53+3

**Language** English

---

### **Abstract**

To support easier integration and better adaptability, Representational State Transfer (REST) Application Programming Interfaces (APIs) are used in many domains. They help system developers keep each functionality and feature isolated and well-maintained at the same time. Within the telecommunication domain, Service-Based Architecture (SBA) exposed the control plane functionality and data repositories of the 5G network with the help of Network Functions (NFs). These NFs used REST APIs to further expose their functionalities; one such example is Network Exposure Function (NEF) API. These APIs are growing exponentially with the addition of new NFs and underlying functionalities. Hence, it started to become cumbersome for system administrators to grasp the technical feasibility of each one of them and invoke the correct API to address user inquiries.

To solve this issue, this thesis demonstrates a fine-tuning experiment on open-source Large Language Models (LLMs) to observe the performance of fine-tuned models in comparison to their non-fine-tuned counterparts as well as one closed-source model, i.e., GPT-4. Although there have been multiple research experiments around the usage of Generative Artificial Intelligence (GenAI) and LLMs to understand and invoke REST APIs, but none of them were focused on specifically telecommunication-related APIs and also not all of them worked around open-source LLMs.

Hence, this thesis fine-tuned two open-source models, namely, Phi-2 and Mixtral, on a dataset consisting of JavaScript Object Notation (JSON) objects of API requests and answers, and evaluated the performance of Phi-2 via two metrics, i.e., GPT4Ref and BertScore, to assess accuracy and similarity of answers with ground truths. The results appeared to be significantly improved in terms of accuracy and similarity with the realization of how effective it can be to use LLMs on domain-specific problems, given the right form of data and sufficient computational resources.

---

**Keywords** Generative AI, Large Language Models, Fine-Tuning, REST APIs, Parameter-Efficient Fine-Tuning , Automation

---

## Acknowledgments

I would like to extend my gratitude to my supervisors, Arto Hellas and Ahmed Hussain, for their invaluable technical guidance and unwavering moral support throughout the course of this thesis. Their constant encouragement and insights have been instrumental in helping me achieve the milestones of this work.

My sincere thanks also go to my team at Ericsson, particularly my manager, Tomas Mecklin, and my advisor, Mukesh Thakur. Their support went beyond technical assistance, as they offered mentorship and encouragement during challenging times. I am especially grateful for the opportunity to work on a topic as innovative as Generative AI, which has allowed me to explore and learn immensely under their guidance.

I am profoundly thankful for the encouragement of my family and friends, especially my mom, dad, and sisters. Despite the distance, they reassured me that I was never alone on this journey, reminding me how fortunate I am to have such a loving family.

Lastly, I would like to express my heartfelt appreciation to my husband for his endless love, care, and kindness. His constant support kept me confident and focused during moments of self-doubt. This journey would not have been possible without his steadfast contribution, always by my side through both the highs and the lows, no matter the distance.

Espoo, 31 October 2024

Zainab Khan

With the support of the  
Erasmus+ Programme  
of the European Union



# Contents

|  |           |
|--|-----------|
| <b>Abstract</b>  | <b>3</b>  |
| <b>Acknowledgments</b>                                     | <b>4</b>  |
| <b>Contents</b>  | <b>5</b>  |
| <b>1 Introduction</b>                                      | <b>8</b>  |
| 1.1 Context and Motivation                                 | 8         |
| 1.2 Problem Description                                    | 9         |
| 1.3 Research Objectives                                    | 9         |
| 1.4 Research Questions                                     | 10        |
| 1.5 Sustainability   | 10        |
| 1.6 Ethical Considerations                                 | 10        |
| 1.7 Scope of Research                                      | 10        |
| 1.8 Structure of the Thesis                                | 11        |
| <b>2 Background and Related Work</b>                       | <b>12</b> |
| 2.1 Web Services and REST APIs                             | 12        |
| 2.2 Large Language Models                                  | 13        |
| 2.2.1 Transformer Model                                    | 14        |
| 2.2.2 Open-Source LLMs                                     | 16        |
| 2.2.3 Adapting LLMs via Retrieval Augmented Generation     | 17        |
| 2.2.4 Adapting LLMs via Fine-Tuning                        | 17        |
| 2.3 Parameter Efficient Fine-Tuning of LLMs for Automation | 19        |
| 2.4 Integrating Large Language Models with APIs            | 20        |
| 2.4.1 RestGPT  | 20        |
| 2.4.2 Gorilla  | 21        |
| 2.4.3 ReST Meets ReAct                                     | 21        |
| 2.4.4 ToolLLM  | 22        |
| <b>3 Methodology and Proposed Solution</b>                 | <b>24</b> |
| 3.1 Requirement Analysis                                   | 24        |
| 3.2 Design and Development                                 | 24        |
| 3.3 Expected Outcome                                       | 25        |
| 3.4 Experimental Setup                                     | 26        |
| 3.4.1 Amazon EC2 Instance                                  | 26        |
| 3.5 Data Generation  | 28        |
| 3.5.1 Generating Synthetic Data with LLM                   | 28        |
| 3.5.2 Data Processing and Scaling                          | 29        |
| 3.6 Fine-Tuning Phi-2                                      | 31        |
| 3.7 Fine-Tuning Mixtral                                    | 32        |

|          |  |           |
|----------|--|-----------|
| <b>4</b> | <b>Evaluation Results</b>  | <b>35</b> |
| 4.1      | Generating Answers from Base Model . . . . .                     | 35        |
| 4.2      | Generating Answers from Fine-tuned Model . . . . .               | 36        |
| 4.3      | Metrics Computation . . . . .                                    | 37        |
| 4.3.1    | GPT-4 Ref Score . . . . .  | 38        |
| 4.3.2    | BertScore . . . . .  | 39        |
| 4.4      | Manual Test of Making API Calls . . . . .                        | 40        |
| <b>5</b> | <b>Analysis and Discussion</b>                                   | <b>41</b> |
| 5.1      | Reflection on Goals . . . . .                                    | 41        |
| 5.2      | Data Format . . . . .  | 41        |
| 5.3      | Prompting LLM . . . . .  | 42        |
| 5.4      | Adapting Fine-tuned Model . . . . .                              | 43        |
| 5.5      | Handling Bigger Models . . . . .                                 | 43        |
| 5.6      | Comparison with Existing Research Work . . . . .                 | 44        |
| 5.6.1    | RestGPT . . . . .  | 44        |
| 5.6.2    | Gorilla . . . . .  | 44        |
| 5.7      | ToolLLM . . . . .  | 44        |
| 5.8      | Limitations and Challenges . . . . .                             | 45        |
| <b>6</b> | <b>Future Work</b>   | <b>46</b> |
| 6.1      | Evaluation of Mixtral and Experimenting with more LLMs . . . . . | 46        |
| 6.2      | Improving the Model for Making API Calls . . . . .               | 46        |
| 6.3      | Security Analysis and Implementation . . . . .                   | 46        |
| 6.4      | Further Automation . . . . .                                     | 47        |
| <b>7</b> | <b>Conclusion</b>  | <b>48</b> |
| <b>A</b> | <b>Prompts used During Thesis</b>                                | <b>54</b> |

## Abbreviations

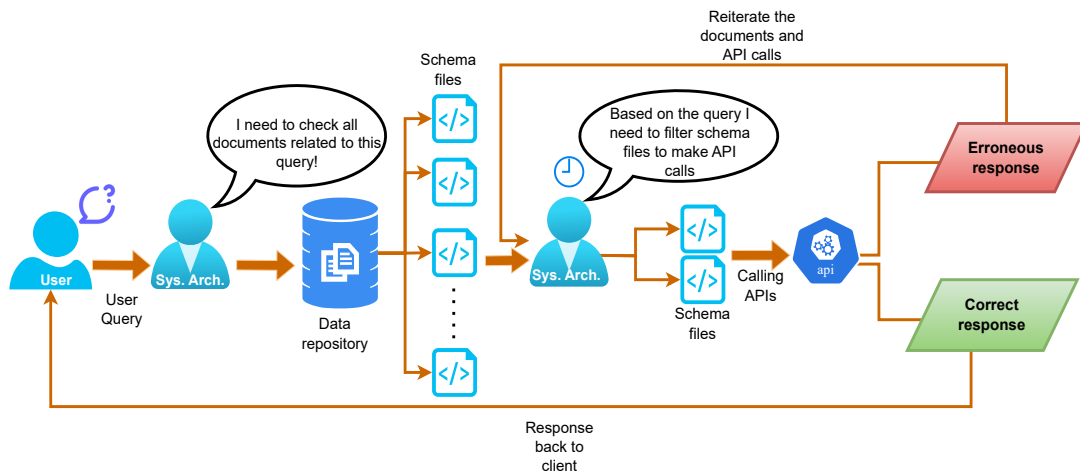
|       |   |
|-------|---|
| AI    | Artificial Intelligence                                 |
| BERT  | Bidirectional Encoder Representations from Transformers |
| EBS   | Elastic Block Store                                     |
| EC2   | Elastic Computing Cloud                                 |
| FFT   | Full Fine-Tuning  |
| GenAI | Generative Artificial Intelligence                      |
| GPT   | Generative Pre-Trained Transformer                      |
| LLM   | Large Language Model                                    |
| LoRA  | Low-Rank Adaptation                                     |
| NEF   | Network Exposure Function                               |
| NF    | Network Function  |
| NLP   | Natural Language Processing                             |
| OAS   | OpenAPI Specification                                   |
| PEFT  | Parameter-Efficient Fine-Tuning                         |
| QLoRA | Q-Low-Rank Adaptation                                   |
| RAG   | Retrieval Augmented Generation                          |
| REST  | Representational State Transfer                         |
| SBA   | Service-Based Architecture                              |
| SBI   | Service-Based Interface                                 |
| 3GPP  | 3rd Generation Partnership Project                      |

# 1 Introduction

## 1.1 Context and Motivation

With the advent of 5G, 3rd Generation Partnership Project (3GPP) defined SBA which supported the delivery of control plane functionality and common data repositories of 5G network via a set of interconnected Network Functions (NFs) [1]. Each of these NFs use Service-Based Interface (SBI) to expose their functionality, where SBI employs REST APIs such as NEF API [2]. Later on, APIs based on micro-service architecture such as CAMARA [3] added more to the complexity involved in navigating APIs across the telecommunication domain. This wide usage and acceptance of APIs in any domain, rather than just telecommunication, gave rise to exposure of products and services via APIs, and each one of them has several different internal and external artifacts such as technical documentation, API endpoints, API schema files, etc.

Although SBI provides several benefits, especially in supporting flexible, resilient, and scalable deployments of NFs, it also introduces complexity due to the increased number of APIs present. This is both due to the compartmentalization of previously monolithic network nodes [4] to micro-services, each having its own APIs, but also due to the introduction of new functionality, that also requires other sets of APIs. This complexity affects both users of these APIs (id est, customers), but also software developers that develop or debug functionality for existing or new types of NFs. Figure 1 illustrates an exemplary process of a user query. The bottleneck is typically a human, in this case, illustrated as a system architect, who analyzes the user query to provide the proper API calls that elicit a response. A simple solution to index these APIs to a database and enable free search. However, that will also be a time-consuming and error-prone approach because knowing the correct API and finding it in a pool of many APIs is cumbersome, and it highly depends on team competence.



**Figure 1:** Traditional flow of answering user query where system architectures has to go through a pool of API artifacts to respond to user inquiries, and meanwhile, there still exists a possibility of making mistakes by invoking wrong APIs.

In this thesis, we explore the possibility of automating this manual process using GenAI and, specifically, LLMs. The idea is to input the relevant API specification documents to the model and train the LLM to produce with the appropriate output. Examples of API specifications that are used as input to the model follow the OpenAPI specification (OAS).

LLMs are pre-trained on Natural Language Processing (NLP)-based tasks and are usually good with general queries, but training them on domain-specific in-context data will be a step up in giving the model an ability to generate accurate and detailed responses. This thesis contributes twofold. First, it reduces the time it takes to process a user query. Second, in future enhancements, it can easily be adapted to generate content other than API calls, such as flow diagrams, code, test cases, etc.

## 1.2 Problem Description

The conventional method described in Section 1.1, which demands significant time and effort from system administrators to respond to user queries, often resulting in potential errors, necessitates a resolution. Another issue is the closed-source nature of foundation models, e.g., OpenAI models [5]. This makes the adaptation of these well-performing models difficult for further enhancement in the field of Artificial Intelligence (AI) and automation, as access to them is limited. Some recent research like RestGPT [6] and ToolLLM [7] have addressed these issues, but none of them used telecommunication datasets.

## 1.3 Research Objectives

In this thesis research project, we aim to design and develop a prototyping system to automate NEF API calls to invoke network functions as per the requirements of user queries. We will be using API specification files of these REST-based NEF APIs which adhere to OAS format as input data to the model.

The implementation will be carried out using open-source LLMs such as Mixtral and Phi-2 (Section 2.2.2). Given the general nature of LLMs, it is assumed that they may have limited telecommunication context and knowledge. To adapt the models for telecommunication tasks, fine-tuning will be performed using a synthetically generated telecommunication dataset from NEF APIs and a Parameter-Efficient Fine-Tuning (PEFT) technique known as Low-Rank Adaptation (LoRA). Further investigation into the training data for these models is warranted to confirm their suitability for telecommunication-related tasks.

The goals of this thesis are outlined as follows:

- To automate the procedure of NEF API calls using OAS-based files.
- To observe the performance of LLMs on domain-specific knowledge with in-context learning, which is not just based on NLP tasks, but rather technical dataset to integrate and communicate with external tools via REST APIs.

- To conduct the experimentation of fine-tuning open-source LLMs, i.e., Mixtral and Phi-2 on domain-specific dataset and compare the results of Phi-2 against Retrieval Augmented Generation (RAG).

## 1.4 Research Questions

The research questions of this thesis are as follows:

- **RQ1:** To what extent does fine-tuning a LLM influence its performance in generating API calls?
- **RQ2:** Can fine-tuned LLMs of reduced complexity achieve similar performance than foundation LLMs in API call generation?

## 1.5 Sustainability

Automating REST API call identification in 5G networks contributes to sustainability by improving efficiency and reducing operational costs. By minimizing manual intervention, it helps save computational resources and energy. However, training and deploying large language models require significant resources, raising concerns about environmental impact. To enhance sustainability, adopting energy-efficient hardware, optimizing training methods, and using renewable energy-powered cloud solutions can help mitigate these challenges.

## 1.6 Ethical Considerations

Using AI for API selection raises ethical concerns related to bias, transparency, and data privacy. Ensuring fairness in model training, maintaining explainability, and complying with data protection regulations are critical for responsible implementation. Additionally, accessibility and resource fairness must be considered to prevent disparities in AI adoption. Promoting open-source models and equitable access to computing resources can help foster ethical AI development.

## 1.7 Scope of Research

The scope of this thesis research is limited to a specific use case, i.e., towards automation of the process to call APIs only using NEF REST APIs dataset. We will be using only two open-source models (i.e., Mixtral and Phi-2) and one PEFT technique, i.e., Q-Low-Rank Adaptation (QLoRA). This can be further extended to test the final trained model against unseen APIs to observe the generalization capabilities of the LLMs but that is out of the scope of this thesis.

## 1.8 Structure of the Thesis

This thesis is organized into 7 chapters. Chapter 2 provides the necessary background context and also presents the relevant literature work revolving around the usage of GenAI for automation of domain-specific tasks. Chapter 3 outlines and explains the approach taken towards implementing this thesis idea and also describes the implementation of the prototype with all technical details. Chapter 4 explains the methodology for evaluating the performance of trained LLMs on a given dataset as well as the results of this evaluation. Chapter 5 gives an overview of our analysis and lessons learned from this thesis. Chapter 6 suggests further expansion of this thesis's findings to automate more time-consuming tasks efficiently. Finally, Chapter 7 concludes this thesis by providing a summary of the entire experiment.

## 2 Background and Related Work

### 2.1 Web Services and REST APIs

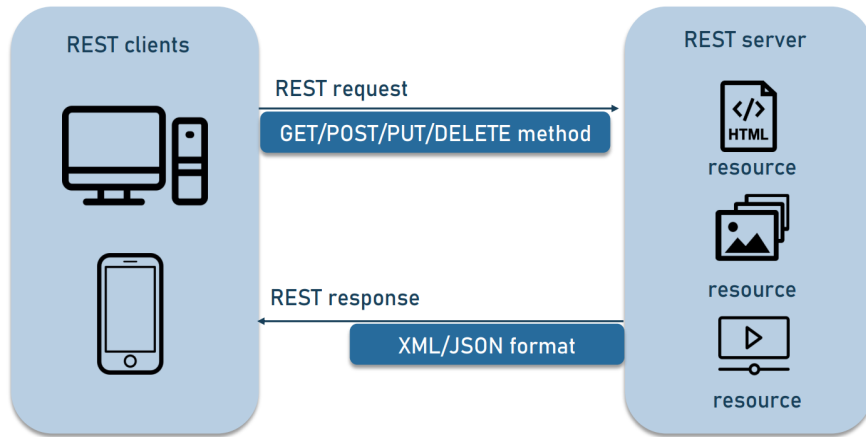
A web service offers a way of communication to two or more electronic devices over a network. It is made available to the clients on the network through a web-based server. Web services are implemented through various protocols such as Simple Object Access Protocol (SOAP) and Representational State Transfer (REST).

REST [8] is an architectural style for distributed systems. It uses API Universal Resource Locators (URLs) to expose the resources, i.e., data, objects, and services. These URLs, also known as endpoints, are used to access and use the data through various Hypertext Transfer Protocol (HTTP) methods. There are five main HTTP methods that are used to perform a variety of operations over exposed resources, those are defined below:

1. **GET:** an HTTP method that is used to read or retrieve information from the database.
2. **POST:** an HTTP method to create a new record in the database.
3. **PUT:** an HTTP method to update the details of a record in the database. It updates (overwrites) all fields of data in the database row and replaces them with the new data.
4. **PATCH:** an HTTP method to also update the details of a record but it only updates the changed data from the new data object into the database row, which means it does not overwrite the entire row of the database.
5. **DELETE:** an HTTP method to delete a record from the database.

REST is based on six main principles which are: uniform interface, decoupling of client and server, statelessness, cacheability, layered system architecture, and code on demand. It offers a way of communication between a server and an HTTP client and uses OpenAPI specification (OAS) [10] to define and document the API. It ensures easier compatibility as both the request and response follow JSON/XML-based schemas. Servers adhering to REST API standards have a base URL and they expose resources through URLs that are relative to the base URL, such as a server can expose an employee database over HTTP and base URL for first version can be '/employee/v1'. The resources will then take relative paths. This is illustrated in Table 1.

There is a wide range of APIs in every domain, however, this thesis is focused on the telecommunication domain utilizing NEF APIs [11, 12]. Within the network domain, the practice of making network functionalities, such as network and data services, accessible to communication service providers is referred to as network or service exposure. This method allows access to network and data resources across various ecosystems while maintaining adherence to security and data integrity policies to foster innovation and enhance enterprise applications [13].



**Figure 2:** Client and Server communication through REST API architecture where the client(s) can request multiple actions on the resources hosted by the server via HTTP methods and the server answers back in JSON/XML format [9]

| HTTP Method | Resource URL  | Description  |
|-------------|---------------|--|
| GET         | /employees    | Returns data of all employees                        |
| GET         | /employees/12 | Returns data of employee#12                          |
| POST        | /employees    | Creates a new employee record in the database        |
| PUT         | /employees/12 | Updates the details of employee#12                   |
| DELETE      | /employees/12 | Removes the details of employee#12 from the database |

**Table 1:** REST API URL examples with different HTTP methods

To cater to the diverse requirements of 5G users, service providers can easily initiate new capabilities and make them accessible through multiple APIs. This enhances the adaptability of connectivity services and their programmability. API exposure in 5G enables developers and third-party applications to access the services and features provided by the 5G network. Additionally, network exposure functions in 5G are designed to reveal specific functionalities and interfaces at a more detailed level, all of which are guided by the 3GPP standardization.

## 2.2 Large Language Models

Large Language Model (LLM) [14] is a Deep Learning (DL) algorithm that comes under the umbrella term of GenAI. With the usage of massive training datasets, LLMs are capable of performing various NLP tasks [15], such as text summarization, translation, generation, and prediction. LLMs are also known as neural networks as they are created by taking the human brain as inspiration. These transformer

architecture-based LLMs (discussed in Section 2.2.1) possess good problem-solving capabilities and are trained to perform a variety of tasks which can benefit different industries such as finance [16], healthcare [17], and entertainment [18].

Some of the key components of a LLM are:

- **Embedding layer** captures the semantic and syntactic meaning of the provided input and creates embeddings, which helps the model understand the context.
- **Feedforward layer (FFN)** helps the LLM to obtain higher-level abstractions from the embeddings to understand the user intent.
- **Recurrent layer** understands and captures the relationship between the words of a sentence.
- **Attention mechanism** provides the model with the ability to focus on the most important part of the input text relevant to the completion of the task at hand.

LLMs are greatly used in different fields and domains these days, however, they also have some challenges such as hallucinations [19], security risks [20], and bias [21], etc.

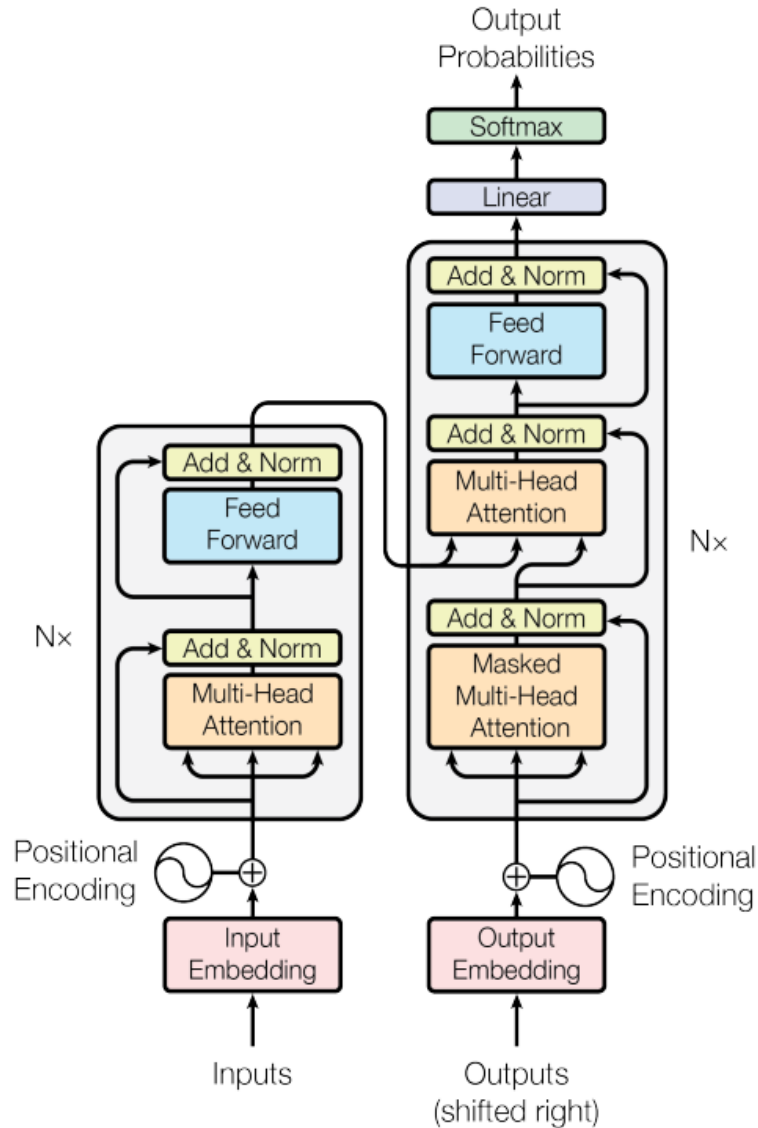
### 2.2.1 Transformer Model

Transformer model [22] is a DL model that is commonly used in various domains, such as NLP tasks and computer vision. It incorporates a self-attention mechanism through which the model can focus on most similar parts of the input to produce each output. Transformers are capable of taking the entire input at once and parallelizing processing, which decreases the training time of the model. They are the base model in the development of large pre-trained models like BERT and GPT [23].

The transformer model is based on transformer architecture (depicted in Figure 3) which consists of two main modules which are the encoder and decoder. This structure does not rely on convolutions and recurrence to generate the output, instead, it uses an attention mechanism that allows modeling of dependencies between input and output sequences without consideration of their distance. This has enabled better parallelization and can achieve a new state-of-the-art level of translation quality.

In this architecture, the encoder does the job of mapping the input sequence represented as  $(x_1, \dots, x_n)$  to a series of continuous representations, i.e.,  $z = (z_1, \dots, z_n)$ , which can then be fed to the decoder. The decoder generates an output sequence represented as  $(y_1, \dots, y_n)$  one element at a time, after receiving the output of the encoder in addition to the decoder's output at a previous step.

This encoder-decoder based architecture converts any input data (mostly strings) into an n-dimensional embedding to be fed into the encoder. Both the encoder and decoder consist of several layers stacked on each other multiple times. Mainly, these are feed-forward and multi-head attention layers.



**Figure 3:** Transformer Model Architecture with encoder and decoder stacks demonstrated along with their underlying layers [22].

**Encoder Stack -** The encoder has 6, i.e.,  $N = 6$  layers stacked on top of each other, and each layer has two sub-layers. Multi-head self-attention mechanism is the first layer and the later is the position-wise feed-forward network. There is a residual connection between every two sub-layers which is followed by layer normalization.

**Decoder Stack -** Similar to the encoder, the decoder also has 6 identical layers. However, the decoder has a third sub-layer in addition to the two sub-layers that the encoder has. This third layer takes the output of the encoder stack and performs multi-head attention over it. Similar to the encoder, it also employs residual connections between layers followed by layer normalization. In the decoder stack the self-attention layer has another modification which ensures that the decoder stack only takes into

account the previous position and not other subsequent positions. This makes it possible that the predictions for position  $i$  can only depend on the outputs from positions less than  $i$ .

### 2.2.2 Open-Source LLMs

In this thesis implementation, the focus is to use open-source LLMs to promote research contribution. While there are plenty of open source LLMs, the discussion here is based on only Phi-2 and Mixtral-8x7b.

#### Phi-2

Phi-2 [24] is a Transformer model of 2.7 billion parameters released by Microsoft Research. It follows the preceding models Phi-1 and Phi-1.5 and is trained on the same data sources which include Python codes, synthetic Python textbooks, and exercise generated by GPT-3.5-turbo-0301 [25], Q&A, and competition codes. In addition to these data sources, Phi-2 has elevated knowledge from various NLP synthetic texts and filtered websites.

Phi-2 has shown comparable performance among models with less than 13 billion parameters when assessed against metrics testing common sense, logical reasoning, and language understanding.

For its training, the architecture of Phi-2 is based on the Transformer model with next word prediction objective. It has a context length of 2048 tokens. The training dataset size was 250B tokens and the training tokens were 1.4T tokens. Phi-2 training took 14 days using 96 A100-80G GPUs. Given the nature of the training data mentioned earlier, Phi-2 is best suited for prompting in Q&A format, chat format, and code format [24].

#### Mixtral-8x7B

The authors in [26] presented Mixtral-8x7b which is licensed under Apache 2.0. It is a sparse mixture of experts model with open weights. The Mixtral-8x7b model consists only of a decoder in which each layer is made up of 8 feed-forward blocks (i.e., experts). Then, at every layer, and for each token, there is a router network that selects two experts from these groups to process the current state of the token and merge their output. In this technique, during inference, each token utilizes only 13B active parameters but has access to 47B parameters. This results in controlling the cost and latency of the model as it only uses a subset of parameters per token.

Mixtral has a context size of 32k tokens and it is pre-trained on multilingual data. Over several benchmarks, Mixtral exhibits either similar performance or sometimes outperforms Llama 2 70B and GPT-3.5. Moreover, in particularly complicated tasks such as code generation and mathematics, it remarkably performs better than Llama 2 70B. Additionally, from its large context window of 32k tokens Mixtral is able to successfully retrieve information regardless of the length of sequence and the location of the information in the sequence.

The authors [26], in addition to Mixtral 8x7b, also presented Mixtral 8x7b-Instruct, which is a fine-tuned chat-model. It uses supervised fine-tuning and direct preference

optimization to follow instructions and performs remarkably well in comparison to other chat models such as Claude-2.1, GPT-3.5 Turbo, Llama 2 70B, and Gemini Pro.

### **2.2.3 Adapting LLMs via Retrieval Augmented Generation**

LLMs are pre-trained on massive multilingual datasets and are capable of handling NLP-based tasks from their existing knowledge. However, sometimes there is a need to equip the LLM with external knowledge to facilitate accurate implementation of domain-specific tasks. To address this, one of the most commonly used approaches is using Retrieval Augmented Generation (RAG). It is the process of providing a LLM with information outside of its training data sources to optimize output generation.

RAG introduces an information retrieval component that uses the input from the user and fetches information related to it from a new data source(s) [27]. This new information along with the actual user query is provided to the LLM, which in turn generates better and more accurate responses.

The following is an overview of how RAG process works:

1. Creating external data outside of the training dataset of the LLM. This data can be fetched from multiple sources and it can have a variety of formats. Then, this data is converted into numerical representations known as embeddings and stored in data stores like vector databases. The knowledge library generated from this step is in a format understandable to GenAI models.
2. In the next step, a relevancy search operation is performed, where the user query is converted into vector representation and matched against the stored vector database. Based on mathematical vector calculations relevant documents to the user query are returned as output of this step.
3. The next step is to construct a prompt for LLM which consists of task description and relevant retrieved data as context for the LLM. There are multiple prompt engineering techniques that help in communicating with LLM effectively. This allows the LLM to generate accurate responses for the user queries.
4. Finally, to overcome the problem of stale data, an ideal RAG model incorporates an asynchronous mechanism to update the documents and embeddings through automated real-time processes or periodic batch processing.

### **2.2.4 Adapting LLMs via Fine-Tuning**

The fine-tuning process [28, 29] takes pre-trained large dataset-based generic models and trains them further on smaller but domain-specific datasets. This improves the performance and accuracy of the model towards fulfillment of a domain-specific task. In other words, fine-tuning takes a general pre-trained model and adapts it to become more suited to the specific needs and expectations of a particular field.

Fine-tuning is a supervised learning approach where the dataset is labeled to meet the requirements of a specific task. This labeled dataset is used to update the weights

of the LLM, and it mostly consists of prompt-response pairs. The following are the main steps of fine-tuning:

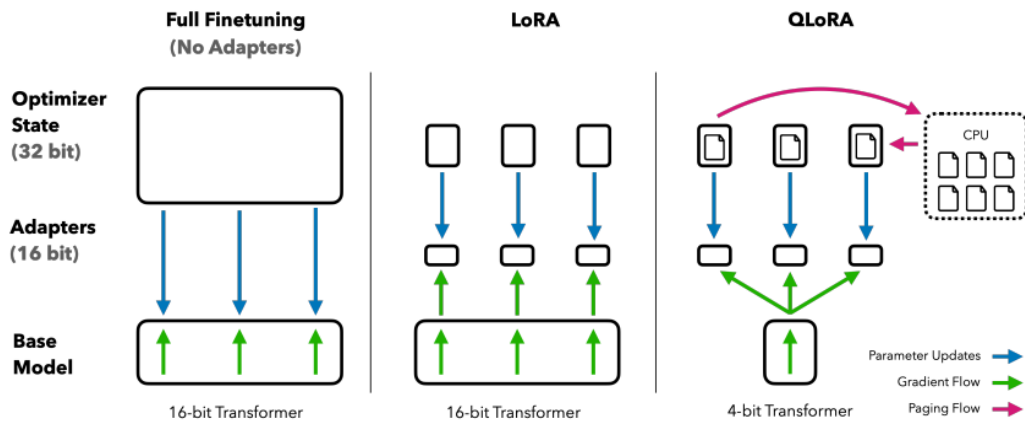
1. The labeled data prepared with a supervised learning approach is divided into training and testing datasets.
2. Some of the prompts are then selected from the training dataset and are passed to the LLM, which then tries to generate completions for those prompts.
3. The model calculates the error or difference between the actual completion and its own prediction.
4. This error rate is then used to adjust weights inside the model through an optimization algorithm. Weights are adjusted according to their contribution towards the error.
5. This weight adjustment is kept ongoing for multiple epochs to reduce the error gradually until the model is transformed enough from general knowledge to task-specific knowledge.
6. During this entire process, the model keeps getting updated with the labeled data, which changes based on the difference between the guesses of the model and the actual answers. Eventually, the model starts exhibiting improved performance for the specific task which it is fine-tuned for.

Mainly, there are two categories of fine-tuning, i.e., Full Fine-Tuning (FFT) and Parameter-Efficient Fine-Tuning (PEFT). In FFT, all of the model's weights are adjusted. This results in a totally new version of the model which is completely different from the actual pre-trained model. Despite the fact that fully fine-tuned models perform well for domain-specific tasks, they end up demanding a great deal of computing resources and memory to update all of the model's weights. They are also prone to challenges like catastrophic forgetting, in which the model loses its ability to cater to any task other than the one it is trained for.

Using FFT approach is a computationally expensive and intensive task. It requires a large amount of storage to store the model in addition to the parameters needed for the training process. This makes it hard to handle with ordinary computers. PEFT comes in handy in this situation, as it provides comparable performance than FFT only by updating a small set of parameters. PEFT algorithms [30] such as LoRA, adapter, prefix tuning, and Masked Head Model (MHM) choose a handful of parameters to change and train on task-specific requirements and freeze the rest of the model parameters.

Among these multiple PEFT algorithms, here, the focus is on using QLoRA through which fine-tuning a quantized 4-bit model is possible without any degradation in performance. It quantizes a pre-trained model using a novel high-precision technique and then attaches a small set of learnable Low-rank Adapter weights that are tuned by backpropagating gradients through quantized weights [31].

QLoRA enables easy accessibility of LLM fine-tuning by reducing the average memory requirements, such that a 65B parameter model which usually requires more than 780GB of GPU memory tones down to demanding less than 48GB of memory without degrading the runtime or predictive performance in comparison to a 16-bit fully fine-tuned baseline model. It does that by introducing 4-bit NormalFloat quantization data type, double quantization, and paged optimizers. PEFT with QLoRA may exhibit slightly degraded performance than FFT as it does not update all of the model weights, however, the benefits of training speed and performance retention outweigh all of these drawbacks. Figure 4 shows a visual representation of how QLoRA improves over LoRA by using paged optimizers to handle memory spikes and quantizing the transformer model to 4-bit precision [31].



**Figure 4:** Full Fine-Tuning vs LoRA vs QLoRA [31].

While PEFT offers better manageability, it also deals with catastrophic forgetting. Since it does not change the model parameters completely, the model does not forget its initially learned knowledge, instead the task-specific knowledge appears to be an additional learning for the model. This way it is also easier, to reuse the same copy of the pre-trained model for different tasks by just modifying a set of parameters according to the nature of the task.

## 2.3 Parameter Efficient Fine-Tuning of LLMs for Automation

As discussed earlier, FFT tends to update all the parameters of the model for each task, and so has proven to be extensively expensive and resource consuming. Considering this, the authors of [30] conducted experiments to observe the results of automating various tasks using PEFT on different state-of-the-art code models. The variety of tasks included defect detection, code clone detection, code summarization, and code translation. They used two pre-trained transformer architecture-based models, i.e., CodeBERT [32] and GraphCodeBERT [33] for code defect and clone detection tasks, however, for code translation and summarization tasks CodeT5 [34] and PLBART [35] were used. Keeping these models, they evaluated results of four PEFT methods, which

include, LoRA, Adapter, Prefix tuning, and MHM, in comparison to FFT. The results of this empirical study in [30] led to the following findings:

1. For code understanding (i.e., code clone and defect detection) tasks PEFT methods exhibited comparable or even better performance than FFT. However, in code generation (code summarization and translation) tasks they showed comparatively weaker performance than FFT.
2. PEFT methods can achieve better performance than FFT in both code understanding and code generation tasks with an increase in the number of trained parameters. However, with further increase in the number of parameters PEFT methods tend to perform lower especially in code understanding tasks.
3. In low-resource scenarios and for code summarization tasks Adapter and LoRA produced better results. Meanwhile, MHM shows better performance in code defect detection task.
4. For code summarization tasks, when updating only 0.5% additional parameters PEFT methods showed comparable performance to FFT.
5. PEFT methods provide better transfer ability between different tasks and projects by leveraging frozen parameters of pre-trained models.

## 2.4 Integrating Large Language Models with APIs

### 2.4.1 RestGPT

RestGPT [6] proposed a framework to connect LLMs with RESTful APIs. This solution consists of three main modules, i.e., planner, API selector, and executor. There is a coarse-to-fine online planning with feedback mechanism incorporated, through which the planner generates several sub-tasks required for the execution, then the API selector maps the sub-task with appropriate API, and finally the executor invokes the API call and receives the results of execution. The executor internally has two sub-modules, a caller and a response parser. The caller makes the API call according to the call parameters provided in the API plan and documentation, then the parser generates python code to parse the response of API call based on the response schema. After receiving the execution results of the sub-task the planner performs online planning of the next subsequent task.

The performance of RestGPT was evaluated using a high-quality human-annotated benchmark dataset, called RestBench, which consisted of two real-world APIs, i.e., the movie database (TMDB) and Spotify music player. The results proved that RestGPT exhibited great performance in generating optimal tasks plan, API selection and invocation, and response parsing in order to handle complex user instructions. RestGPT's performance was also evaluated against various other models and the results suggest that the framework achieved 75% success rate for movie database and 70% for the music player.

### 2.4.2 Gorilla

Gorilla [36] presented a LLM that is connected with extensive APIs. They used self-instruction fine-tuning and retrieval-aware training to allow the LLMs carry out a certain task with usage of correct provided from a big and evolving dataset. The dataset consisted of Machine Learning (ML) APIs from TorchHub, TensorHub, and HuggingFace. They generated the synthetic instruction dataset with GPT-4 to provide in-context examples. After which, they fine-tuned Gorilla which is a LLaMA-7B-based model with document retriever technique which enables the model to keep up with the changes in the API documentation.

Gorilla performed better than GPT-4 with respect to API functionality as well as exhibited very reduced hallucination errors. Gorilla supports two inference modes, i.e., zero-shot and with retrievers. The retrievers used in the retrieval mode are BM25 and GPT-Index. To evaluate the model performance, and see if the LLM is calling right API from the dataset, they used Abstract Syntax Tree (AST) tree-matching strategy. Their results concluded that the finetuned Gorilla model with appropriate retriever performed better than prompting state-of-the-art GPT-4 LLM in addition to reducing hallucination and errors.

### 2.4.3 ReST Meets ReAct

The authors in [37] implemented a ReAct-style LLM agent which can reason and act according to external knowledge. They also improved the agent through a ReST-like approach in which the agent iteratively trains itself on previous trajectories. This approach employs growing-batch reinforcement learning incorporating AI-based feedback which results in continuous self-improvement and self-distillation. They designed a search agent which uses web search to generate long-form answers for diverse open-ended questions. The search agent works as follows:

1. **Incoming question:** agent receives the question and start searching for it.
2. **Decision step:** then it decides if there is a need for additional information to generate appropriate answer. If yes, it summarizes the current information and restart the search, otherwise terminates the search loop.
3. **Answer generation:** agent generates the first draft of the answer based on collected information so far.
4. **Relevance self-check:** then the agent performs two self-revision calls to produce final answer. One of the calls verifies the relevancy of answer to the actual question, while the second one checks that the answer is grounded in retrieved snippets.

To implement this search agent they constructed few-shot prompts for every reasoning step that the agent takes and the prompts were formatted as Python code. PaLM- 2 [38] base models of different sizes were used to run the search agent, and internal Google Q&A API was used as the search tool. Each completed search agent

trajectory was splitted into reasoning steps and FFT was applied. These fine-tuning samples are ranked using a reward model to select best high-scoring samples for fine-tuning of the model. Finally, for evaluation they chose Bamboogle [39] dataset which consisted of 125 2-hop questions. To judge accuracy of the model, they introduces LLM-based auto-eval (built upon base model of PaLM 2-L) and compared the alignment of results with human ratings on Bamboogle trajectories. To further verify that the model is not overfitting the small dataset of Bamboogle only, they used another small test dataset of 100 questions and called it BamTwoogle. BamTwoogle exclusively measured the performance of final model and it was also constructed keeping the consideration that every question requires 2+ steps to be answered.

#### 2.4.4 ToolLLM

The authors of ToolLLM [7] emphasized on the fact that due to the focus of current instruction tuning on only basic language tasks open-source LLMs like LLaMA [40] exhibit limited performance in tool-use capabilities to carry out tasks which require to invoke external APIs for execution. However, closed-source LLMs like ChatGPT [41] and GPT-4 [42] have proven to provide great results in such scenarios but their closed-source nature keeps the inner functionality opaque to the world. To bridge this gap, they introduced ToolLLM that is open-source and works as a general tool-use framework encompassing data construction, model training, and evaluation of results. They also released an instruction-tuning dataset known as ToolBench [43] which is constructed via ChatGPT. The construction of ToolBench mainly comprised of three main phases:

1. **API collection:** using RapidAPI Hub [44] 16,464 RESTful APIs spanning 49 diverse categories were collected. For each API detailed documents were gathered which included API parameters, description, code snippets, etc. These crucial documents can help the LLM in learning execution of APIs as well as generalizing unseen APIs.
2. **Instruction generation:** a subset of APIs is chosen to prompt ChatGPT [41] for generating diverse instructions incorporating scenarios of both single and multi-tool usage. The prompt for ChatGPT consisted of: (1) general description of the expected instruction, (2) detailed documentation of APIs related to the instruction, (3) three in-context examples, i.e., seeds written by human experts.
3. **Solution path annotation:** coming up with a working solution path for a particular instruction is cumbersome for LLMs in practical scenarios, as it may need multiple rounds of reasoning and API calls. To address this, the authors used Depth-First Search-based Decision Tree (DFSDT) which allows the LLMs to evaluate possible solution paths and then either choosing a promising path or abandoning a non-functional path while expanding a new path. This approach worked efficiently to cater both simple as well as complex instructions.

To evaluate results of ToolBench they developed an automatic evaluator based on ChatGPT, known as, ToolEval. It had two evaluation metrics, i.e., pass rate

and win rate, and through rigorous testing it was observed that ToolEval achieved alignment of 87.1% pass rate and 80.3% win rate in comparison to human annotators. The final results of ToolLLaMA (fine-tuned version of ToolBench) demonstrated remarkable performance in handling single-tool as well as multi-tool instructions. It outperformed Text-Davinci-003 and Claude-2, and showed comparable results against ChatGPT. Additionally, the generalization abilities of this tool were remarkable on out-of-distribution dataset, i.e., APIBench, in comparison to Gorilla [36] which is trained on APIBench.

## 3 Methodology and Proposed Solution

### 3.1 Requirement Analysis

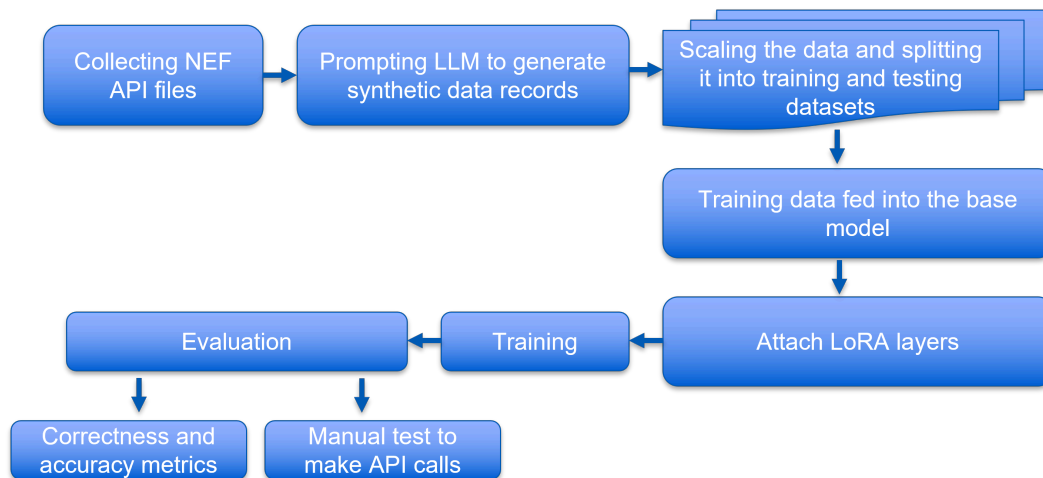
To automate the traditional workflow discussed in Section 1.2, the following requirements should be met:

1. Usage of open-source LLMs to promote further extension and improvement in research.
2. Providing LLMs comprehensive in-context knowledge about NEF APIs to deal with end-users inquiries effectively.
3. Effectively creating the optimal dataset from NEF API specification files for the comprehension of model.
4. Experimenting with at least two open-source LLMs of varying sizes for better understanding and comparison of the output generated by them.

To meet these requirements, we will fine-tune Phi-2 and Mixtral with a specific dataset obtained from NEF API specification file. Additionally, we will evaluate the performance of the fine-tuned model against RAG-based setup on GPT-4.

### 3.2 Design and Development

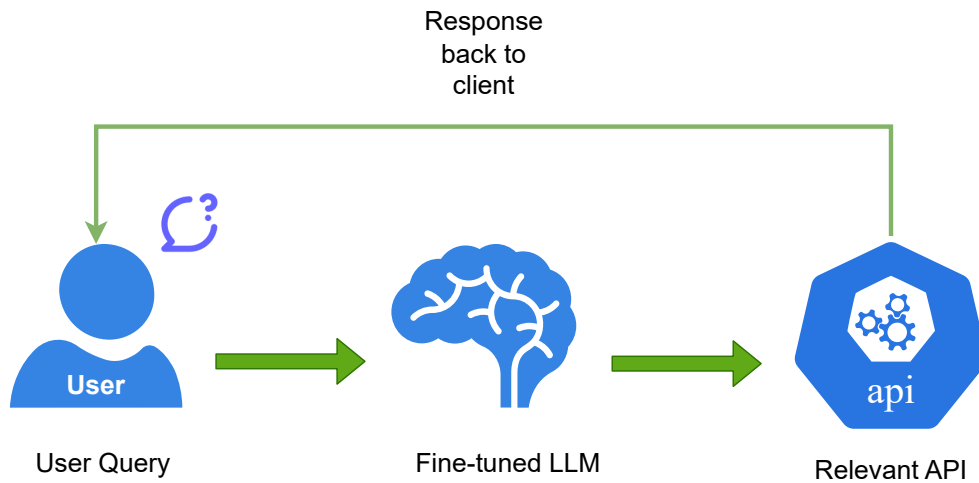
This thesis is focused on the design and implementation of a pipeline of interconnected processes, each producing an output to serve as input for the subsequent step in the sequence. The pipeline, depicted in Figure 5, encompasses the collection of the dataset and the solicitation of a teacher model to generate data records in a specific format from it, the scaling of the generated data, the partitioning of the data into training and testing subsets, the provision of the training dataset to the open-source LLM with QLoRA layers added on top, the initiation of the training process, and finally the assessment of the trained model using the testing dataset.



**Figure 5:** Implementation pipeline of the experiment consisting of multiple processes such as synthetic data generation and scaling, fine-tuning the LLMs and their evaluation.

### 3.3 Expected Outcome

The outcome of this thesis experiment will be specifically trained models that take user queries as input and, after analyzing the technical aspects of the query, will initiate the necessary API call to the relevant API from a selection of multiple APIs as depicted in Figure 6. This will ultimately reduce the time-consuming efforts currently expected from a technical person (such as a system administrator or system architect), as illustrated in Figure 1. Additionally, this research has the potential to pave the way for automating various other aspects of managing systems and products to align with client needs by leveraging the automation capabilities provided by LLMs. Finally, since this experiment is entirely based on open-source technologies, it is easier to customize to a greater extent while also maintaining a controlled check on production costs without compromising data privacy.



**Figure 6:** Automated Flow with the usage of fine-tuned LLMs where the traditional procedure of going through API specification files and invoking the required API to answer user inquiries is simplified by just prompting the trained model to perform the search and invocation of API.

### 3.4 Experimental Setup

To experiment and evaluate the presented methodology, powerful Graphics Processing Units (GPUs) that enable parallel execution and offer sufficient memory and storage are needed. These considerations led us to use Amazon Elastic Computing Cloud (EC2) G5 instances for all experiments.

#### 3.4.1 Amazon EC2 Instance

The G5 instances used in the implementation of this thesis belong to the latest generation of NVIDIA GPU-based instances and are widely used for resource-intensive tasks such as machine learning inference. They are proven to deliver 3.3x higher performance than Amazon EC2 G4dn instances for machine learning training [45]. For data generation, training, and testing of Phi-2, we used the g5.4xlarge instance, and the specifications of the g5.4xlarge instance are listed in Table 2.

| <b>Instance Property</b> | <b>Value</b>  |
|--------------------------|---------------|
| No# of GPUs              | 1             |
| GPU memory               | 24 GiB        |
| Virtual CPUs             | 16            |
| Memory                   | 64 GiB        |
| Storage                  | 1x600         |
| Network bandwidth        | Up to 25 Gbps |
| EBS bandwidth            | 8 Gbps        |
| On-demand price per hour | \$1.624       |

**Table 2:** Details of Amazon EC2 g5.4xlarge Instance [45]

To perform the fine-tuning of Mixtral, a machine with significantly larger memory and storage capacity is needed, as well as more GPUs. The base model of Mixtral Instruct-8x7b, chosen for this experiment, alone required approximately 87 Gigabytes (GBs) of storage space, leading us to anticipate that the fine-tuned model would be even larger. In order to accurately estimate the space required for the fine-tuned model, we initially fine-tuned the model to save and load only the best checkpoint, revealing an estimated size of approximately 15 GBs per checkpoint. Given that we intended to store a checkpoint after every 10 training steps, as indicated in Table 6 (i.e., save steps = 10), a machine capable of accommodating all checkpoints (90 to be specific) of Mixtral’s fine-tuning was necessary. To meet these demands, we utilized the g5.48xlarge instance from the G5 collection, as all smaller instances were either unable to provide sufficient CUDA memory or lacked the necessary storage capacity. However, the storage of this instance was still insufficient to store all the checkpoints, prompting us to attach an additional 1.5 Terabytes (TBs) of Elastic Block Store (EBS) to address this limitation. Equipped with this powerful machine and supplementary storage, we were ultimately able to fine-tune Mixtral, with the complete model consuming approximately 967 GBs and each checkpoint occupying 11 GBs—closely aligning with our initial estimations derived from the earlier best checkpoint of the model. The specifications of Amazon’s g5.48xlarge instance are detailed in Table 3.

| <b>Instance Property</b> | <b>Value</b>   |
|--------------------------|----------------|
| No# of GPU(s)            | 8              |
| GPU memory               | 192 GiB        |
| Virtual CPUs             | 192            |
| Memory                   | 768 GiB        |
| Storage                  | 2x38000        |
| Network bandwidth        | Up to 100 Gbps |
| EBS bandwidth            | 19 Gbps        |
| On demand price per hour | \$16.288       |

**Table 3:** Details of Amazon EC2 g5.48xlarge Instance [45]

## 3.5 Data Generation

In order for LLMs to effectively incorporate a precise understanding of telecommunication APIs within their contextual knowledge, it is necessary to fine-tune them using a specific dataset. In the context of this thesis, NEF API files [13] were utilized for this purpose. To achieve this, we leveraged the API specification YAML files of NEF APIs to create synthetic data. The data generation, processing, and scaling are described below.

### 3.5.1 Generating Synthetic Data with LLM

Initially, we utilized the original dataset, which consisted of the YAML specification files of NEF APIs. To facilitate the model’s comprehension, we flattened one of the YAML files, as each API specification file contains references to other files, with no external references beyond the file itself.

Next, we formulated a prompt for the LLM to generate synthetic data in JSON format, encompassing specific fields such as request, API call, description, method, operation, and parameters. Each of these fields is explained in Table 4.

| Field name  | Description   |
|-------------|---|
| Request     | A question about utilizing the NEF API to perform a specific action.                          |
| Api call    | The full URL of the API endpoint being invoked.   |
| Description | A brief summary of the purpose of the API endpoint.   |
| Method      | The HTTP method used to call the API endpoint.  |
| Operation   | The operationId of the API endpoint.  |
| Parameters  | Any parameters required by the API endpoint, a comma-separated dictionary of key-value pairs. |

**Table 4:** Fields included in each JSON object of the dataset

For this task of synthetic data generation, we employed GPT-4 [42] as the expert model to ensure the highest quality of data. Although the API specification delineates schema definitions for seven API endpoints and the prompt specified the return of solely real data, the LLM (GPT-4) produced some fabricated and unreal JSON objects. The prompt for data generation can be found in Listing 7. Listing 1 illustrates the appearance of the generated JSON data.

```

1 {
2   "request": "How can I obtain an access token for future requests
3   ?",
4   "api_call": "/api/v1/login/access-token",
5   "description": "OAuth2 compatible token login, get an access
6   token for future requests",
7   "method": "post",
8   "operation": "login_access_token_api_v1_login_access_token_post
9   ",
10  "parameters": {
11    "grant_type": "password",
12    "username": "string",
13    "password": "string",
14    "scope": "string",
15    "client_id": "string",
16    "client_secret": "string"
17  }
18 },
19 {
20   "request": "How can I read active subscriptions?",
21   "api_call": "/api/v1/3gpp-as-session-with-qos/v1/{scsAsId}/
22   subscriptions",
23   "description": "Get subscription by id",
24   "method": "get",
25   "operation": "
26     read_active_subscriptions_api_v1_3gpp_as_session_with_qos_v1
27     __scsAsId__subscriptions_get",
28   "parameters": {
29     "scsAsId": "string"
30   }
31 },

```

**Listing 1:** Generated synthetic JSON data consisting of request to invoke certain API and the fields needed to make the call.

Subsequently, we progressed to the data-cleaning phase. Here, we manually reviewed the dataset generated by GPT-4 and eliminated every JSON object containing hallucinated data. Ultimately, we retained precisely seven authentic JSON objects.

### 3.5.2 Data Processing and Scaling

After obtaining only seven records of data from the previous step, we recognized the inadequacy of this dataset for effective model training. Consequently, we opted to prompt the GPT-4 model once more to expand the dataset. To accomplish this, we supplied the model with the request field of each JSON record and requested the LLM to generate 100 unique variations of each request. This process resulted in the generation of 765 records of data. The prompt for scaling the data is given in Listing 8.

Subsequently, we partitioned this expanded dataset roughly into a 70/30 ratio for training and evaluation purposes, resulting in approximately 535 records in the training dataset and 230 in the evaluation dataset. To facilitate the model training, we converted the JSON data into Comma Separated Values (CSV) format, as it was deemed suitable for the model trainer's comprehension. Additionally, for Phi-2, we structured each data record into *Instruct* – *Output* pairs. The *Instruct* field contained the value from the request parameter, while the *Output* object encompassed the remaining parameters such as API call, description, method, operation, and parameters. For

Mixtral, each data record is enclosed in `< s >< /s >` tags, and the request is within the `[INST][/INST]` tags. The structure of such records in the CSV file is illustrated in Listings 2 and 3. The complete process of data generation is depicted in Figure 7.

```
<s>Instruct: What is the method to create a new subscription?\, Output
: {api_call: /api/v1/3gpp-as-session-with-qos/v1/{scsAsId}/
subscriptions\, description: Create Subscription\, method: post\,
operation:
create_subscription_api_v1_3gpp_as_session_with_qos_v1__scsAsId_
_subscriptions_post\, parameters: {'scsAsId': 'string'}}</s>

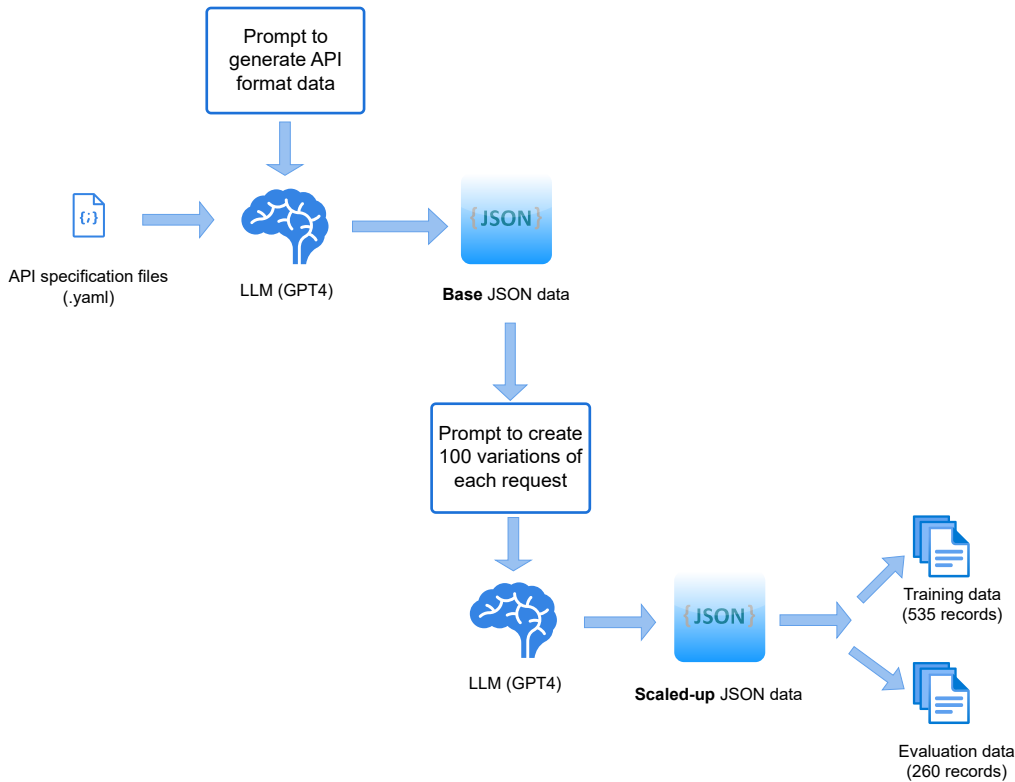
<s>Instruct: Fetch a certain subscription using its ID\, Output: {
api_call: /api/v1/3gpp-as-session-with-qos/v1/{scsAsId}/
subscriptions/{subscriptionId}\, description: Get subscription by
id\, method: get\, operation:
read_subscription_api_v1_3gpp_as_session_with_qos_v1__scsAsId_
_subscriptions__subscriptionId_get\, parameters: {'scsAsId': 'string
'\, 'subscriptionId': 'string'}}</s>
```

**Listing 2:** CSV data generated from the JSON objects of data for Phi-2.

```
<s>[INST] What is the method to create a new subscription? [/INST]
api_call: /api/v1/3gpp-as-session-with-qos/v1/{scsAsId}/
subscriptions\, description: Create Subscription\, method: post\,
operation:
create_subscription_api_v1_3gpp_as_session_with_qos_v1__scsAsId_
_subscriptions_post\, parameters: {'scsAsId': 'string'}</s>

<s>[INST] Fetch a certain subscription using its ID [/INST] api_call:
/api/v1/3gpp-as-session-with-qos/v1/{scsAsId}/subscriptions/{
subscriptionId}\, description: Get subscription by id\, method:
get\, operation:
read_subscription_api_v1_3gpp_as_session_with_qos_v1__scsAsId_
_subscriptions__subscriptionId_get\, parameters: {'scsAsId': 'string
'\, 'subscriptionId': 'string'}</s>
```

**Listing 3:** CSV data generated from the JSON objects of data for Mixtral.



**Figure 7:** Synthetic data generation process consisting of providing NEF API specification file in YAML format to GPT-4 and prompting it to generate question and answer pairs from it, then prompting the model to create 100 variations of each record and finally splitting the generated dataset into 70/30 ratio to constitute training and testing dataset.

### 3.6 Fine-Tuning Phi-2

As discussed in Section 2.2.2, Phi-2 is a Transformer-based small model of 2.7 billion parameters, trained on Python textbook and coding data in addition to NLP based synthetic texts and filtered websites for education purposes. In this thesis, we chose Phi-2 for fine-tuning on telecommunication-based API dataset for mainly observing its performance based on the size of the model and the open-source nature of Phi-2.

The fine-tuning process of Phi-2 consisted of the following steps:

1. Formatting of the training dataset by providing it proper labeling and enclosing each record in *Instruct/Output* format as shown in Listing 2
2. After data formatting, we loaded the CSV dataset file into the system.
3. Next, we loaded the base model of Phi-2 (non-fine-tuned version) without any quantization but with FlashAttention-2, which optimizes the attention mechanism of the model for longer sequences by improving the computational efficiency and reducing memory usage.

4. After loading the base model, we tokenized it and applied the QLoRA configuration on it in addition to training arguments. These QLoRA configuration and training arguments are listed in Table 5 and 6.
5. Finally, we provided the model, QLoRA configuration, and training arguments to Supervised Fine-Tuning Trainer (SFT) [46] from HuggingFace.

The fine-tuning of the Phi-2 model demonstrated computational efficiency, with a training runtime of approximately 595 seconds. The model processed 4.495 training samples per second and 1.504 training steps per second, with a total Floating-Point Operations (FLOs) of  $6.08 \times 10^{15}$ . The final training loss for this model was 0.1921, indicating a reasonable balance between computational efficiency and model performance. These metrics suggest that Phi-2 is well-suited for scenarios where faster fine-tuning is desirable, even if the final loss is slightly higher compared to alternative models. Table 7 provides a summary of these training logs.

| Argument Name  | Argument Value                          |
|----------------|---|
| LoRA alpha     | 16                                      |
| LoRA dropout   | 0.1                                     |
| LoRA rank      | 64                                      |
| Target modules | q_proj, k_proj, v_proj, dense, fc1, fc2 |
| Bias           | None                                    |
| Task type      | CAUSAL_LM                               |

**Table 5:** QLoRA Configuration Arguments

| Argument Name               | Argument Value     |
|-----------------------------|--------------------|
| Train runtime               | 595.1469           |
| Training samples per second | 4.495              |
| Training steps per second   | 1.504              |
| Total flos                  | 6077030969333760.0 |
| Training loss               | 0.1920796508       |

**Table 7:** Training Logs of Phi-2

### 3.7 Fine-Tuning Mixtral

This thesis also explores the fine-tuning of another open-source LLM on telecommunication data, namely Mixtral 8x-7b-Instruct [26], a fine-tuned chat-model version of Mixtral. Unlike Phi-2 [24], Mixtral is a relatively large model, comprising approximately 47B tokens, and has been trained on multilingual data.

The fine-tuning procedure for Mixtral closely parallels that of Phi-2, as detailed in Section 3.6, with the exception that the data is formatted such that the question

| Argument Name               | Argument Value    |
|-----------------------------|-------------------|
| Epochs                      | 5                 |
| Batch size                  | 3                 |
| Gradient accumulation steps | 1                 |
| Optim                       | paged_adamw_32bit |
| Save steps                  | 10                |
| Logging steps               | 10                |
| Learning rate               | $2e^{-4}$         |
| Weight decay rate           | 0.001             |
| Warmup ratio                | 0.03              |
| BF16                        | True              |
| Max grad norm               | 0.3               |
| Max steps                   | -1                |
| Group by length             | True              |
| Scheduler type              | Constant          |
| Reports to                  | Tensorboard       |

**Table 6:** Training arguments provided to SFT from HuggingFace to start fine-tuning of the model.

is enclosed within `[INST][/INST]` tags, and the entire record of question and answer is encompassed within `< s >< /s >` tags. This formatting of the dataset, shown in Listing 3, is implemented to enhance the model’s comprehension of each record’s question and answer while appropriately distinguishing them across all data records [47]. Subsequently, the succeeding steps mirror those of the fine-tuning process for Phi-2, including loading the base model without quantization using FlashAttention-2, tokenizing it, and applying the QLoRA configuration, as well as the training arguments detailed in Table 5 and 6. Notably, the target modules for Mixtral within the training arguments encompass `q_proj`, `k_proj`, `v_proj`, `o_proj`, `w1`, `w2`, and `w3`. Finally, the model was trained using SFT.

In contrast to Phi-2, Mixtral required significantly more time for fine-tuning, with a training runtime of 6945 seconds. The throughput was lower, processing 0.385 training samples per second and 0.129 training steps per second, with FLOs amounting to  $9.29 \times 10^{16}$ . However, the final training loss was 0.1619, slightly lower than Phi-2’s. This indicates that Mixtral prioritizes precision over computational efficiency, making it better suited for tasks where achieving a lower loss is critical, even at the expense of longer training times. The training logs are presented in Table 8.

| <b>Argument Name</b>        | <b>Argument Value</b> |
|-----------------------------|-----------------------|
| Train runtime               | 6944.5925             |
| Training samples per second | 0.385                 |
| Training steps per second   | 0.129                 |
| Total flos                  | 9.285266355e+16       |
| Training loss               | 0.1619244023          |

**Table 8:** Training Logs of Mixtral

## 4 Evaluation Results

For evaluating the performance of fine-tuned open-source LLMs, we drew a comparison between the base model and its fine-tuned version. This comparison is based on two matrices that measure the accuracy and similarity between the answers from both models. To start with, we had already separated testing data during the data generation process as discussed in Section 3.5. From a total of 765 records of data, a sub-portion of 230 records are used as evaluation data. This dataset consists of the following fields:

- **Question:** this field corresponds to a user query about NEF API, that potentially requires identifying the relevant API call to make.
- **Ground truth:** this is the answer from the expert model for the given query. In this case, the model being used as an expert is GPT-4 [42].
- **Result/Answer:** this field represents the answer produced by the model under evaluation (i.e., base model or fine-tuned model) for a given query.

To evaluate the consistency of the results for each model, we ran the evaluation scripts twenty-five times. This also provided us with a clear understanding of the analyzing, understanding, and output capabilities of the base and fine-tuned models. During each evaluation run, three JSON files are generated, which include prediction data, BertScore 4.3.2, and GPT-4 Ref Score 4.3.1 files. The details of each of these files will be discussed in subsequent sections of this chapter.

### 4.1 Generating Answers from Base Model

To derive responses from the primary Phi-2 model, we employed an approach based on the RAG framework. This decision was made due to the base model lacking specific knowledge about the dataset, necessitating the provision of relevant documents as contextual input. To achieve this, we first loaded the YAML file containing the API specification, then segmented it into meaningful chunks using LangChain's recursive text splitter [48]. Subsequently, we generated embeddings using HuggingFace and implemented a similarity search retriever with the Facebook AI Similarity Search (FAISS) library from LangChain [49, 50]. Finally, we integrated LangChain's Q&A [51] with the model, incorporating the provided prompt detailed in Listing 9.

The output produced by the Phi-2 model is then saved in a JSON file, alongside the corresponding question and the ground truth (obtained from Generative Pre-trained Transformer (GPT)-4). The response generated by the base model of Phi-2 is stored under the key labeled "result". A representation of the prediction data is presented in Listing 4, illustrating that the result from Phi-2, without undergoing fine-tuning, lacks proper formatting and does not convey meaningful information.

```

1 {
2   "question": "How do I acquire information on a particular QoS
3     subscribed session?",
4   "ground_truth": {
5     "api_call": "/api/v1/3gpp-as-session-with-qos/v1/{scsAsId}/
6       subscriptions/{subscriptionId}",
7     "description": "Get subscription by id",
8     "method": "get",
9     "operation": "
10       read_subscription_api_v1_3gpp_as_session_with_qos_v1__scsAsId__
11       subscriptions__subscriptionId__get",
12     "parameters": {
13       "scsAsId": "string",
14       "subscriptionId": "string"
15     }
16   },
17   "result": "\n{\n  api_call: '/api/v1/3gpp-as-session-with-qos/v1/{
18     scsAsId}/subscriptions': {'get': {'tags': ['Session With QoS API
19     '], 'summary': 'Read Active Subscriptions', 'description': 'Get
20     subscription by id', 'operationId':
21     read_active_subscriptions_api_v1_3gpp_as_session_with_qos_v1_
22     _scsAsId__subscriptions_get', 'parameters': [{'required': True, '
23     schema': {'title': 'The ID of the Netapp that creates a
24     subscription', 'type': 'string'}, 'example': 'myNetapp', 'name': '
25     scsAsId', 'in': 'path'}]}, 'responses': {'200': {'description': '
26     Successful Response', 'content': {'application/json': {'schema':
27     {'$ref': '#/components/schemas/AsSessionWithQoSSubscription'}}}},
28     '422': {'description': 'Validation Error', 'content',
29     'latency': 7.018583799
30 }

```

**Listing 4:** Prediction data generated by Phi-2 before fine-tuning consisting of JSON object of question, ground truth, result, and latency.

## 4.2 Generating Answers from Fine-tuned Model

In contrast to the base model, the fine-tuned Phi-2 model is capable of addressing NEF API-related inquiries without requiring supplementary context. This thesis aims to substantiate this hypothesis. To achieve this, we assessed the fine-tuned version of Phi-2, employing the same dataset of 230 records (outlined in Section 3.5.2), without the inclusion of additional contextual documents. Loading the fine-tuned model involved simply attaching the tokenizer with the same task-specific prompt as the base model of Phi-2 to the trained model and utilizing the HuggingFace pipeline for loading. The code snippet depicting the pipeline parameters is presented in Listing 5.

```

1 pipeline(task = "text-generation", model = <path-to-fine
2   -tuned-model>, tokenizer = phi2-tokenizer,
3   max_new_tokens = twenty-five6, torch_dtype = torch.
4   bfloat16, trust_remote_code = True, device = 0,
5   temperature = float(0.1), do_sample = True, top_k =
6   10, num_return_sequences = 1, pad_token_id =
7   tokenizer.eos_token_id)

```

**Listing 5:** HuggingFace Pipeline for Loading Fine-Tuned Phi-2

The output from the fine-tuned model was initially in string format, even though in the prompt specified in the appendix, we instructed the model to provide the output in JSON format. Consequently, through manipulation and formatting of the string response generated by the fine-tuned Phi-2, we successfully produced a valid JSON structure, as the response contained all the necessary fields. The ultimate entry, denoted by the key "answer," was then stored in the file alongside the respective question and ground truth, resembling the example depicted in Figure 6.

```

1  {
2  {
3  "question": "How do I acquire information on a particular QoS
4  subscribed session?",
5  "ground_truth": {
6  "api_call": "/api/v1/3gpp-as-session-with-qos/v1/{scsAsId}/
7  subscriptions/{subscriptionId}",
8  "description": "Get subscription by id",
9  "method": "get",
10 "operation": "
11     read_subscription_api_v1_3gpp_as_session_with_qos_v1__scsAsId_
12     _subscriptions__subscriptionId_get",
13 "parameters": {
14 "scsAsId": "string",
15 "subscriptionId": "string"
16 }
17 },
18 "answer": {
19 "api_call": "/api/v1/3gpp-as-session-with-qos/v1/{scsAsId}/
20 subscriptions/{subscriptionId}",
21 "description": "Get subscription by id",
22 "method": "get",
23 "operation": "
24     read_subscription_api_v1_3gpp_as_session_with_qos_v1__scsAsId_
25     _subscriptions__subscriptionId_get",
26 "parameters": {
27 "scsAsId": "string",
28 "subscriptionId": "string"
29 }
30 }
31 },
32 "latency": 12.28957574
33 }

```

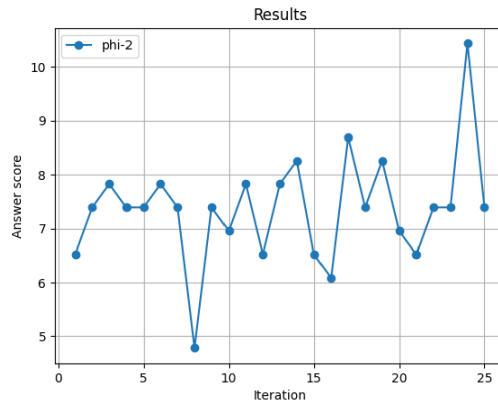
**Listing 6:** Prediction data generated by Phi-2 after fine-tuning consisting of JSON object of question, ground truth, answer, and latency.

### 4.3 Metrics Computation

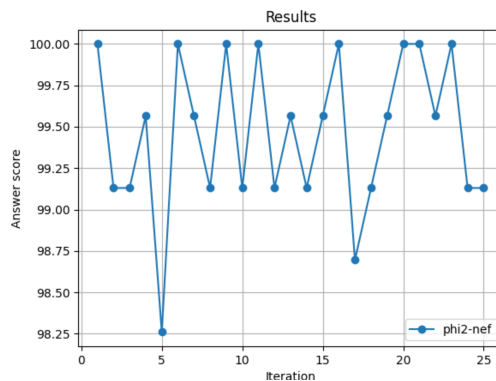
To assess the quality of responses generated by the base and fine-tuned models, we applied two metrics to the prediction data. This dataset comprises twenty-five JSON files containing data records, illustrated in Figure 4 for the base model and Figure 6 for the fine-tuned model. Our evaluation employed the GPT-4 Ref Score and BertScore metrics to measure the accuracy and similarity between the ground truth and the responses from the fine-tuned model. Each metric was executed for twenty-five iterations, resulting in twenty-five score values per metric. Detailed explanations of these metrics are provided in Section 4.3.1 and 4.3.2, and Table 9 gives a summary on the results from the evaluation iterations.

### 4.3.1 GPT-4 Ref Score

We utilized the GPT-4 [42] model to evaluate and validate the accuracy of responses generated by the fine-tuned model in comparison to the ground truth produced by GPT-4, serving as the expert model. To conduct this assessment, we employed GPT-4 Ref Score [52] metric which uses LangChain's QAEvalChain [53]. The scores for the base model of Phi-2 demonstrated an accuracy range of 4.78 to 10.43 on a scale of 0-100 within this evaluation matrix. In contrast, for the fine-tuned version, the scores ranged between 98.69 and 100.00. This considerable disparity demonstrates that the accuracy of language models can undergo substantial enhancement when trained for a specific task with appropriate datasets and parameters. The graphs illustrating all twenty-five evaluations within this matrix are presented in Figure 8 and 9.



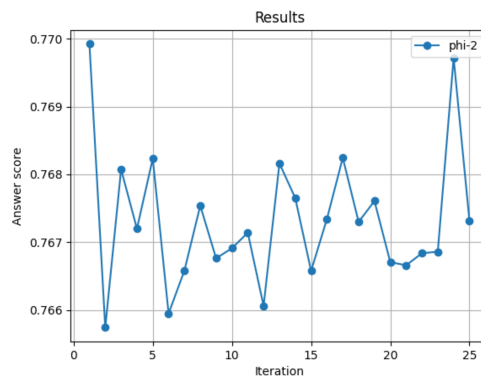
**Figure 8:** GPT-4 Ref matrix evaluation on Phi-2 **before** fine-tuning where the graph is plotted against the number of iterations and accuracy score of the answer. There are a total of 25 iterations, and the accuracy range is from 0-100. However, the resulting values of the model before fine-tuning range from only 4 to 10.



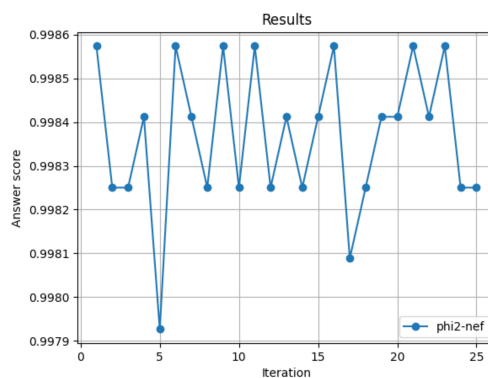
**Figure 9:** GPT-4 Ref matrix evaluation on Phi-2 **after** fine-tuning where the graph is plotted against the number of iterations and accuracy score of the answer. There are a total of 25 iterations, and the accuracy range is from 0-100. However, the resulting values of the model after fine-tuning range from 98 to 100.

### 4.3.2 BertScore

The second matrix utilized for assessment in this thesis is BertScore [54]. This metric quantifies the resemblance between the response generated by the fine-tuned Phi-2 and the actual correct response. In this methodology, the natural language of both the ground truth and the generated answer is transformed into vector embeddings and subsequently compared using a pairwise cosine-similarity approach. The BertScore for Phi-2 falls within the range of 0.765 and 0.769 on a scale of 0-1 to gauge similarity. However, for the fine-tuned version of Phi-2, the range is between 0.997 and 0.998. The graphs illustrating both models can be found in Figures 10 and 11. Once again, it is evident that the similarity score has improved in the fine-tuned version.



**Figure 10:** BertScore matrix evaluation on Phi-2 **before** fine-tuning where the graph is plotted against the number of iterations and similarity score of the answer. There are a total of 25 iterations, and the similarity range is from 0-1. However, the resulting values of the model before fine-tuning range from 0.7657 to 0.7699



**Figure 11:** BertScore matrix evaluation on Phi-2 **after** fine-tuning where the graph is plotted against the number of iterations and similarity score of the answer. There are a total of 25 iterations, and the similarity range is from 0-1. However, the resulting values of the model after fine-tuning ranges from 0.9979 to 0.9986

|                           | GPT-4 Ref Score |          | BertScore |          |
|---------------------------|-----------------|----------|-----------|----------|
|                           | Phi-2           | Phi2-NEF | Phi-2     | Phi2-NEF |
| <b>Maximum</b>            | 10.4348         | 100      | 0.7699    | 0.9986   |
| <b>Minimum</b>            | 4.7826          | 98.2609  | 0.7657    | 0.9979   |
| <b>Median</b>             | 7.3913          | 99.4609  | 0.7673    | 0.9984   |
| <b>Standard Deviation</b> | 1.0027          | 0.4647   | 0.0009    | 0.0003   |

**Table 9:** The Maximum, Minimum, Median, and Standard Deviation values calculated on GPT-4 Ref Score and BertScore values obtained during the twenty-five evaluation runs on Phi-2 and fine-tuned Phi-2

#### 4.4 Manual Test of Making API Calls

Thus far, the fine-tuned model has demonstrated exceptional proficiency in generating accurate responses for queries related to the NEF API dataset, surpassing the performance of Phi-2’s base model. However, in order to manually evaluate the fine-tuned model, we conducted actual API calls to the NEF server.

To accomplish this, we loaded the fine-tuned model and provided it to LangChain’s create openapi agent [55] along with basic testing credentials. This agent, based on the OAS, is designed to comprehend user queries, devise an execution plan, and subsequently execute the necessary web or API requests. Despite the fine-tuned Phi-2’s proficiency in generating answers, it did not exhibit satisfactory performance when integrated with the agent. The outputs produced were consistently random and lacked coherence. Several factors, such as integration issues with the agent or the relatively limited size of Phi-2, may contribute to this behavior. Further investigation into these factors and refinement of the model are earmarked for future research.

In this chapter, we have extensively explored the enhancements achieved through fine-tuning the base model of Phi-2, demonstrating that even a small LLM can exhibit exceptional performance when trained for a specific task using appropriate datasets and training parameters. Additionally, we fine-tuned Mixtral for the purpose of this thesis experiment to observe the impact of model size on the generated results. However, due to the substantial size of Mixtral, as detailed in Section 3.4.1, we were unable to assess the fine-tuned model of Mixtral, as it necessitated a single GPU with greater memory capacity than that provided by the g5.48xlarge instance. Currently, our infrastructure did not include machines with larger capabilities than those found in the G5 collection. Consequently, the evaluation of Mixtral will be a potential area for future exploration in this thesis.

## 5 Analysis and Discussion

### 5.1 Reflection on Goals

The main goals of this thesis, as outlined in Section 1.3, are to automate the process of NEF API calls through open-source models and to evaluate their performance after fine-tuning on a domain-specific dataset. To achieve this, we applied Parameter-Efficient Fine-Tuning (PEFT) to two open-source LLM, specifically Phi-2 and Mixtral. The results showed a significant performance improvement in Phi-2 following fine-tuning. However, the fine-tuning process made the models extremely large, making storage and inference challenging. Due to limited resources, we were unable to evaluate the fine-tuned Mixtral model. Nonetheless, the experiments provided valuable insights and helped us address the research questions raised in Section 1.4. Below, we list the previously introduced research questions and how they are addressed:

- **RQ1:** How much fine-tuning an LLM can influence its performance in the task of generating API calls?

**Answer:** Fine-tuning an LLM on a domain-specific dataset can significantly enhance its performance. For example, the initial accuracy score of the non-fine-tuned model was between 4-10 out of 100. After fine-tuning, accuracy scores increased to 98-100, with similarity scores improving from 0.7 to 0.9 on a scale of 1.

- **RQ2:** Can fine-tuned language models of reduced complexity have similar performance with foundation LLM counterparts in the task of API call generation?

**Answer:** Yes. We evaluated both GPT-4 Ref Score and BertScore metrics against the ground-truth answers generated by the GPT-4 model and the scores from the fine-tuned model demonstrated performance very close to that of the teacher model, confirming that even models with reduced complexity can achieve similar results after fine-tuning.

### 5.2 Data Format

During fine-tuning of an LLM, it is very important to take sufficient time to decide the data format that you want your model to learn from. Undoubtedly, LLMs have proven to be very smart and adaptable to a variety of tasks. However, a high probability exists for them to learn and get trained on the wrong data format and lose direction.

During this thesis, the format of data kept changing continuously until we successfully trained the LLM to understand, analyze, and compute the specific format of output required for the use case. Table 10 highlights the data formats explored during this experiment and the observations out of them, i.e., the type, structure, and amount of records of data for training and testing according to the nature of the task and the libraries used.

| Data format                              | Observation   |
|--|---|
| Question and Answer                      | The Q&A dataset suits the best for normal user queries out of textual data, but for REST APIs, this data format proved to be insufficient. In short, the model was unable to suggest relevant API call based on a user query.   |
| JSON (Question and ground truth) objects | Next, we constructed JSON objects of question and answer pairs, where the question corresponds to the user query and the answer object contains multiple fields of API data (i.e., API call, description, method, operation, and parameters) as shown in Listing 1. Training the model with this data did not work either because we used SFTTrainer [46] from HuggingFace, and it required data to be in plain text format.  |
| Unlabeled CSV data                       | Next, we converted the JSON objects into CSV data while merging all of the fields inside the answer object and provided it to SFTTrainer as plain text data. The model was trained on this dataset without any technical errors, but it lacked the sense of constructing the desired form of output object with all the required fields. The model produced outputs by combining random fields from the dataset, as without proper labeling of fields, it could not differentiate between their values. |
| Labeled CSV data                         | Finally, we enclosed each record of the CSV file with Instruct and Output labels as shown in Listing 2 for Phi-2 and in [INST][/INST] tags for Mixtral as shown in Listing 3. Additionally, we assigned labels to each field, such as API call, description, etc. After training the LLM on this dataset, it was finally able to provide the correct API data as output along with proper identification and differentiation among fields of the JSON object.   |

**Table 10:** Variations of data formats including question and answer, JSON objects, unlabeled CSV data, and labeled CSV data.

### 5.3 Prompting LLM

Each LLM interprets user instructions and expectations differently based on the task requirements, such as whether the task necessitates a conversational or question-and-answer approach. Effective prompting is essential for ensuring that the selected model accurately comprehends the intricacies of the task. Initially, in our experiment, we informally prompted Phi-2 [24] without specific formatting, which did not yield the optimal output. Subsequently, we employed a question-and-answer format for

prompting [56] Phi-2, as the model appeared to better understand this particular structure, incorporating the keywords "Instruct" to denote the task or query and "Output" to indicate the expected result or answer. However, in the case of Mixtral, we used tags, i.e., `[INST][/INST]` to enclose the question and `< s >< /s >` tags to enclose the rest of the prompt [47].

## 5.4 Adapting Fine-tuned Model

The focus of this thesis experiment was to achieve optimal outcomes for a task specific to a particular domain and use case, and the results thus far have sufficiently supported our hypothesis. Nevertheless, concerns arise regarding the applicability of a fine-tuned model to a similar task using a different dataset. Key questions include whether it is feasible to apply this fine-tuned LLM to another REST API specification, and if so, how this could be accomplished. While these questions have not been addressed or experimented within the scope of this thesis, a hypothetical solution might involve integrating a RAG component with the fine-tuned LLM.

Given that the model is already proficient in analyzing and understanding OAS-based API specifications, its capabilities could be further evaluated using an alternative API dataset in a RAG-based setup to achieve comparable results and performance efficiency. This process would entail dividing flattened specification files into chunks using an appropriate text splitter, creating embeddings from these chunks, and then utilizing a retriever to facilitate the model's inference process.

This approach could serve as an intriguing experiment to assess the adaptability of a fine-tuned model to datasets distinct from those used in its initial training. However, such an investigation currently falls outside the scope of this thesis.

## 5.5 Handling Bigger Models

One of the most critical considerations when training any LLM is ensuring the availability of sufficient computational resources. It is important to take into account the size of the LLM to be trained and then estimate the potential size of the fine-tuned model. The models can become quite large when fine-tuned with smaller step sizes and if all generated checkpoints are to be stored. Handling these LLMs necessitates larger machines, greater memory, and increased storage capacity. In order to reduce the size of these LLMs, there are options to either increase the step size, which would result in less frequent storage of checkpoints, or to only store the best checkpoint. However, these solutions come with their own trade-offs, such as potential degradation in the quality of responses. We have provided a more detailed discussion of these computational requirements for Phi-2 and Mixtral in Section 3.4.1.

## 5.6 Comparison with Existing Research Work

### 5.6.1 RestGPT

RestGPT [6] focused on connecting LLMs with RESTful APIs, which is very similar to the motivation of this thesis. However, they designed and developed a prompt-based framework of three main modules, i.e., planner, API selector, and executor, which is supposed to integrate with any REST API to understand the task, decompose it into actions and select the correct API. The main difference between RestGPT and our experiment is that we essentially targeted APIs from the telecommunication domain, and we used open-source models for fine-tuning instead of only using prompts. RestGPT, on the other hand, used text-davinci-003 from OpenAI for all of its implementation. Another notable difference is that this thesis is more focused on assessing fine-tuning results on open-source models in comparison to the closed-source counterparts, making the solution more accessible and usable for further research. Finally, their success rates on the selected datasets for choosing and making correct API calls is 70-75%, whereas, the prediction accuracy of the solution proposed by this thesis is 98-100% given that we did not experiment with multiple domains, rather than just telecommunication.

### 5.6.2 Gorilla

Gorilla [36] is a very close work that has similar motivation and research questions as this thesis. They presented a fine-tuned LLM connected with extensive APIs. Two main similarities between this thesis and Gorilla are: (i) Fine-Tuning: we used Parameter-Efficient Fine-Tuning (PEFT) and Gorilla used self-instruction fine-tuning with retrieval aware training, and (ii) Open-source LLMs: Gorilla used an open-source LLaMA-7B-based model to fine-tune and develop their solution. Additionally, for evaluation, they compared their results against GPT-4, and Gorilla performed better than GPT-4 with respect to API functionality, as well as exhibited very reduced hallucination errors. However, in this thesis, we used GPT-4 as the teacher model, and the fine-tuning results were only compared against the non-fine-tuned counterparts of the same model but not against GPT-4.

## 5.7 ToolLLM

The research work of ToolLLM [7] is also inspired by the same thought as this thesis, which is to utilize the open-source LLMs to design solutions and frameworks that can communicate with RESTful APIs to bridge the communication gap with external knowledge. ToolLLM offers a complete framework encompassing the functionalities of encompassing data construction, model training, and evaluation of results. However, their API dataset is collected from RapidAPI Hub spanning 49 diverse categories, unlike a single-domain focused approach taken in this thesis.

## 5.8 Limitations and Challenges

The work done during this thesis highlighted the importance of fine-tuning LLMs when there is a domain-dedicated task requiring automation. However, it also made us realize the limitations that fine-tuning models can bring to the experiment. Some of those are listed below.

- Fine-tuning models on dedicated datasets and tasks may limit their performance for generic jobs, especially FFT.
- Fine-tuning LLMs require a substantial amount of computing resources. Not only does the fine-tuning process itself take a huge amount of memory during the model training, but also the storage of fine-tuned models can be very challenging. Depending on the number of model checkpoints that you choose to store of the fine-tuned model, the resulting model keeps getting bigger, and if you are already working with a LLM with billions of parameters, the fine-tuned version of it with all the checkpoints can get too huge to fit in a single machine or storage system. Hence, it is important to have an estimate of the required resources for fine-tuning the model, storing it, and running inference on it. This challenge was faced during this thesis when we trained Mixtral and ran out of memory to run inference on it for its evaluation.
- For evaluation of the models, this thesis calculated two metrics, i.e., GPT-4 Ref Score and BertScore, to measure accuracy and similarity between the generated answer and the ground truth answer. However, there can be more metrics to assess the performance of the LLMs, such as a metric to compute the hallucinations made by the model, etc.
- Once a model is fine-tuned, it becomes an expert for the knowledgebase used during its training. However, this can be problematic once that data becomes stale. To combat this situation, one possibility is to combine both fine-tuning and RAG-based approaches. This will allow the model to use its training information to provide answers to known queries or otherwise use the latest external knowledge with the help of RAG setup, such an approach is taken by Gorilla [36].
- Lastly, this thesis utilized a small dataset of JSON objects obtained from the NEF API file. Training on such a limited dataset may lead to overfitting. Therefore, using a comparatively larger dataset in the future could significantly impact the performance.

## 6 Future Work

### 6.1 Evaluation of Mixtral and Experimenting with more LLMs

In this thesis, we have completed the fine-tuning of both Phi-2 and Mixtral models, along with the evaluation of Phi-2. However, the evaluation of Mixtral remains incomplete due to the unavailability of the necessary computing infrastructure. The model expanded (in size) significantly after fine-tuning, reaching 967 GBs, necessitating a machine with larger GPUs than the g5.48xlarge, which was the largest available machine in our case. Therefore, evaluating the performance of fine-tuned Mixtral is the most prioritized aspect of this thesis that requires future attention.

Additionally, we observed that after fine-tuning, the performance of Phi-2 incredibly improved with regards to predicting the right API call; however, as discussed in Section 4.4, the model was still unable to make a complete API call which requires connecting to the NEF server and executing the request. For this aspect, we aim to conduct this experiment with further improvement in chosen parameters and available resources to fine-tune more models in the future. To begin with, Llama models, Phi-3, and Mistral are the future candidates to observe the performance on planning and execution of actual API calls to the NEF server.

### 6.2 Improving the Model for Making API Calls

As outlined in Section 4.4, it was observed that Phi-2, after fine-tuning, did not effectively execute actual API calls when integrated with create OpenAPI agent [55]. This highlights the need for further investigation into enhancing the model training to enable seamless integration with intelligent agents for planning, strategizing, and executing one or more API calls to the server.

### 6.3 Security Analysis and Implementation

The use of GenAI and LLMs technologies has demonstrated significant potential for automating a wide range of tasks. This thesis has confirmed the hypothesis that appropriately trained LLMs can achieve high performance and low error rates in automating domain-specific tasks, such as providing accurate NEF API calls in response to user queries. However, LLMs are vulnerable to security risks and can be fooled [57]. They are prone to many attacks such as instruction backdoor [58], prompt injection [59], and jailbreak attack [60, 61].

Hence, conducting a thorough security analysis of LLM-based automation before implementing real-world solutions is essential. Identifying and addressing potential security vulnerabilities through appropriate measures is paramount to ensure that the final solution is sufficiently secure for integration into enterprise systems and applications.

## **6.4 Further Automation**

Upon consideration of the aforementioned aspects, this solution may be extended to automate additional tasks that typically demand human attention and time. One potential enhancement involves training a LLM to generate code for a specified functionality of an API call in the desired programming language. However, while this implementation and model training are beyond the scope of this thesis, they represent an intriguing avenue for automating a comprehensive pipeline with the aid of LLMs.

## 7 Conclusion

To conclude, within the scope of this thesis, the main idea was to determine the practical feasibility of automation that can be brought by the usage of GenAI and LLMs. To address the objectives, we automated the procedure of calling NEF API calls with open-source LLMs. The motivation to do so was to observe the capabilities of LLMs in understanding not just NLP-based tasks but also technical ones, especially REST API specification files.

To do so, we created a synthetic dataset of JSON objects from the OAS-based API files and fine-tuned Phi-2 and Mixtral on the dataset. For fine-tuning, we chose QLoRA as PEFT technique and finally compared the performance of the mentioned models before and after fine-tuning. We experimented with multiple formats of dataset (i.e., NEF API), prompting techniques, and computation resources to achieve better results. The results showed a significant difference in performance between models accompanied with RAG and a fine-tuned version of the model. The accuracy score of answers went from the values of 4-10 to 98-100, and the similarity or resemblance of answers with ground truth answers increased from 0.7 to 0.9, answering the first research question of this thesis, which is about observing the impact of fine-tuning on the performance of LLMs.

This significant improvement in results of fine-tuned models on our API dataset helped us in addressing the second research question of this thesis, that is if LLMs are trained on dedicated tasks provided with the appropriate dataset, they can produce as accurate results as their foundation models counterparts without much human intervention. This leads us to the possible extension of this thesis and enables the freedom to use LLMs in more tasks involved in the software development lifecycle, such as system design, creating required APIs, generating code, testing, and deployment, etc. The possibility of automation on such enhanced grounds can greatly reduce the time consumption and efforts needed from technical experts as well as it may produce less erroneous results.

## References

- [1] L. Xia, M. Zhao, and Z. Tian, “5g service based core network design,” in *2019 IEEE Wireless Communications and Networking Conference Workshop (WCNCW)*, 2019, pp. 1–6.
- [2] Network exposure - enable 5g innovation. [Online]. Available: <https://www.ericsson.com/en/core-network/network-exposure>
- [3] J. Ordonez-Lucena and F. Dsouza, “Pathways towards network-as-a-service: the camara project,” in *Proceedings of the ACM SIGCOMM Workshop on Network-Application Integration*, ser. NAI '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 53–59. [Online]. Available: <https://doi.org/10.1145/3538401.3546825>
- [4] J. van Zyl, “A perspective on service based architecture,” in *Proceedings of SAICSIT*, vol. 249, 2002.
- [5] “Openai.” [Online]. Available: <https://openai.com/>
- [6] Y. Song, W. Xiong, D. Zhu, W. Wu, H. Qian, M. Song, H. Huang, C. Li, K. Wang, R. Yao, Y. Tian, and S. Li, “Restgpt: Connecting large language models with real-world restful apis,” 2023.
- [7] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, L. Hong, R. Tian, R. Xie, J. Zhou, M. Gerstein, D. Li, Z. Liu, and M. Sun, “Toollm: Facilitating large language models to master 16000+ real-world apis,” 2023.
- [8] V. B. Surwase, “Rest api modeling languages - a developer’s perspective,” *International Journal For Science Technology And Engineering*, vol. 2, pp. 634–637, 2016. [Online]. Available: <https://api.semanticscholar.org/CorpusID:54773316>
- [9] “Rest api: Key concepts, best practices, and benefits.” [Online]. Available: [https://www.altexsoft.com/media/2021/03/rest\\_api\\_works.png](https://www.altexsoft.com/media/2021/03/rest_api_works.png)
- [10] “Openapi specification.” [Online]. Available: <https://swagger.io/specification/>
- [11] J. d. Gregorio, “jdegre/5gc\_apis,” original-date: 2018-05-30T11:18:36Z. [Online]. Available: [https://github.com/jdegre/5GC\\_APIs](https://github.com/jdegre/5GC_APIs)
- [12] Swagger editor. [Online]. Available: [https://jdegre.github.io/editor/?url=https://raw.githubusercontent.com/jdegre/5GC\\_APIs/master/TS29591\\_Nnef\\_EventExposure.yaml](https://jdegre.github.io/editor/?url=https://raw.githubusercontent.com/jdegre/5GC_APIs/master/TS29591_Nnef_EventExposure.yaml)
- [13] “Network Exposure - Enable 5G Innovation.” [Online]. Available: <https://www.ericsson.com/en/core-network/network-exposure>

- [14] “What is a large language model (llm)?” [Online]. Available: <https://www.elastic.co/what-is/large-language-models>
- [15] L. Qin, Q. Chen, X. Feng, Y. Wu, Y. Zhang, Y. Li, M. Li, W. Che, and P. S. Yu, “Large language models meet nlp: A survey,” 2024. [Online]. Available: <https://arxiv.org/abs/2405.12819>
- [16] H. Zhao, Z. Liu, Z. Wu, Y. Li, T. Yang, P. Shu, S. Xu, H. Dai, L. Zhao, G. Mai, N. Liu, and T. Liu, “Revolutionizing finance with llms: An overview of applications and insights,” 2024. [Online]. Available: <https://arxiv.org/abs/2401.11641>
- [17] Z. A. Nazi and W. Peng, “Large language models in healthcare and medical domain: A review,” *Informatics*, vol. 11, no. 3, 2024. [Online]. Available: <https://www.mdpi.com/2227-9709/11/3/57>
- [18] M. Schatten, “AI and the Future of Entertainment Technology,” Jul. 2024, working paper or preprint. [Online]. Available: <https://hal.science/hal-04637685>
- [19] Z. Xu, S. Jain, and M. Kankanhalli, “Hallucination is inevitable: An innate limitation of large language models,” 2024. [Online]. Available: <https://arxiv.org/abs/2401.11817>
- [20] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang, “A survey on large language model (llm) security and privacy: The good, the bad, and the ugly,” *High-Confidence Computing*, vol. 4, no. 2, p. 100211, 2024. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S266729522400014X>
- [21] I. O. Gallegos, R. A. Rossi, J. Barrow, M. M. Tanjim, S. Kim, F. Deroncourt, T. Yu, R. Zhang, and N. K. Ahmed, “Bias and Fairness in Large Language Models: A Survey,” *Computational Linguistics*, vol. 50, no. 3, pp. 1097–1179, Sep. 2024, eprint: [https://direct.mit.edu/coli/article-pdf/50/3/1097/2471010/coli\\_a\\_00524.pdf](https://direct.mit.edu/coli/article-pdf/50/3/1097/2471010/coli_a_00524.pdf). [Online]. Available: [https://doi.org/10.1162/coli\\_a\\_00524](https://doi.org/10.1162/coli_a_00524)
- [22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: [https://proceedings.neurips.cc/paper\\_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf)
- [23] N. Heidloff. Foundation models, transformers, BERT and GPT. [Online]. Available: <https://heidloff.net/article/foundation-models-transformers-bert-and-gpt/>
- [24] microsoft/phi-2 · hugging face. [Online]. Available: <https://huggingface.co/microsoft/phi-2>

- [25] “Openai. models gpt-3.5.” [Online]. Available: <https://platform.openai.com/docs/models/gpt-3-5>
- [26] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. de las Casas, E. B. Hanna, F. Bressand, G. Lengyel, G. Bour, G. Lample, L. R. Lavaud, L. Saulnier, M.-A. Lachaux, P. Stock, S. Subramanian, S. Yang, S. Antoniak, T. L. Scao, T. Gervet, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, “Mixtral of experts,” 2024.
- [27] “What is RAG? - Retrieval-Augmented Generation Explained - AWS.” [Online]. Available: <https://aws.amazon.com/what-is/retrieval-augmented-generation/>
- [28] C. Jeong, “Domain-specialized llm: Financial fine-tuning and utilization method using mistral 7b,” *Journal of Intelligence and Information Systems*, vol. 30, no. 1, p. 93–120, Mar. 2024. [Online]. Available: <http://dx.doi.org/10.13088/jiis.2024.30.1.093>
- [29] B. Zhang, Z. Liu, C. Cherry, and O. Firat, “When scaling meets llm finetuning: The effect of data, model and finetuning method,” 2024. [Online]. Available: <https://arxiv.org/abs/2402.17193>
- [30] J. Liu, C. Sha, and X. Peng, “An empirical study of parameter-efficient fine-tuning methods for pre-trained code models,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 397–408.
- [31] T. Detrmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, “QLoRA: Efficient Finetuning of Quantized LLMs,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 10 088–10 115, Dec. 2023. [Online]. Available: [https://proceedings.neurips.cc/paper\\_files/paper/2023/hash/1feb87871436031bdc0f2beaa62a049b-Abstract-Conference.html](https://proceedings.neurips.cc/paper_files/paper/2023/hash/1feb87871436031bdc0f2beaa62a049b-Abstract-Conference.html)
- [32] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “Codebert: A pre-trained model for programming and natural languages,” 2020.
- [33] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, “Graphcodebert: Pre-training code representations with data flow,” 2021.
- [34] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” 2021.
- [35] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “Unified pre-training for program understanding and generation,” 2021.
- [36] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, “Gorilla: Large language model connected with massive apis,” 2023.

- [37] R. Aksitov, S. Miryoosefi, Z. Li, D. Li, S. Babayan, K. Kopparapu, Z. Fisher, R. Guo, S. Prakash, P. Srinivasan, M. Zaheer, F. Yu, and S. Kumar, “Rest meets react: Self-improvement for multi-step reasoning llm agent,” 2023.
- [38] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, S. Shakeri, E. Taropa, P. Bailey, Z. Chen, and et al., “Palm 2 technical report,” 2023.
- [39] “Bamboogle.” [Online]. Available: <https://paperswithcode.com/dataset/bamboogle>
- [40] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, “Llama: Open and efficient foundation language models,” 2023.
- [41] “Openai. chatgpt, 2022.” [Online]. Available: <https://openai.com/blog/chatgpt>
- [42] “Openai. gpt-4 and gpt-4 turbo.” [Online]. Available: <https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo>
- [43] “Toolbench.” [Online]. Available: <https://github.com/OpenBMB/ToolBench>
- [44] “Rapid api hub.” [Online]. Available: <https://rapidapi.com/hub>
- [45] Amazon EC2 g5 instances | amazon web services. [Online]. Available: <https://aws.amazon.com/ec2/instance-types/g5/>
- [46] “Supervised Fine-tuning Trainer.” [Online]. Available: [https://huggingface.co/docs/trl/en/sft\\_trainer](https://huggingface.co/docs/trl/en/sft_trainer)
- [47] Mixtral – nextra. [Online]. Available: <https://www.promptingguide.ai/models/mixtral#prompt-engineering-guide-for-mixtral-8x7b>
- [48] Recursively split by character | LangChain. [Online]. Available: [https://python.langchain.com/v0.1/docs/modules/data\\_connection/document\\_transformers/recursive\\_text\\_splitter/](https://python.langchain.com/v0.1/docs/modules/data_connection/document_transformers/recursive_text_splitter/)
- [49] Faiss: A library for efficient similarity search. [Online]. Available: <https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/>
- [50] Faiss | LangChain. [Online]. Available: <https://python.langchain.com/v0.2/docs/integrations/vectorstores/faiss/>
- [51] Q&a with RAG | LangChain. [Online]. Available: [https://python.langchain.com/v0.1/docs/use\\_cases/question\\_answering/](https://python.langchain.com/v0.1/docs/use_cases/question_answering/)
- [52] A. Karapantelakis, M. Thakur, A. Nikou, F. Moradi, C. Olrog, F. Gaim, H. Holm, D. D. Nimara, and V. Huang, “Using large language models to understand telecom standards,” in *2024 IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN)*, 2024, pp. 440–446.

- [53] QAEvalChain — LangChain documentation. [Online]. Available: [https://api.python.langchain.com/en/latest/langchain/evaluation/langchain.evaluation.qa.eval\\_chain.QAEvalChain.html](https://api.python.langchain.com/en/latest/langchain/evaluation/langchain.evaluation.qa.eval_chain.QAEvalChain.html)
- [54] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, “Bertscore: Evaluating text generation with bert,” 2020. [Online]. Available: <https://arxiv.org/abs/1904.09675>
- [55] create\_openapi\_agent — LangChain documentation. [Online]. Available: [https://python.langchain.com/v0.2/api\\_reference/community/agent\\_toolkits/langchain\\_community.agent\\_toolkits.openapi.base.create\\_openapi\\_agent.html](https://python.langchain.com/v0.2/api_reference/community/agent_toolkits/langchain_community.agent_toolkits.openapi.base.create_openapi_agent.html)
- [56] Phi-2 – nextra. [Online]. Available: <https://www.promptingguide.ai/models/phi-2#phi-2-usage>
- [57] S. Abdali, J. He, C. Barberan, and R. Anarfi, “Can llms be fooled? investigating vulnerabilities in llms,” 2024. [Online]. Available: <https://arxiv.org/abs/2407.20529>
- [58] R. Zhang, H. Li, R. Wen, W. Jiang, Y. Zhang, M. Backes, Y. Shen, and Y. Zhang, “Instruction backdoor attacks against customized LLMs,” in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1849–1866. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-rui>
- [59] Y. Liu, G. Deng, Y. Li, K. Wang, Z. Wang, X. Wang, T. Zhang, Y. Liu, H. Wang, Y. Zheng, and Y. Liu, “Prompt injection attack against llm-integrated applications,” 2024. [Online]. Available: <https://arxiv.org/abs/2306.05499>
- [60] J. Chu, Y. Liu, Z. Yang, X. Shen, M. Backes, and Y. Zhang, “Comprehensive assessment of jailbreak attacks against llms,” 2024. [Online]. Available: <https://arxiv.org/abs/2402.05668>
- [61] X. Guo, F. Yu, H. Zhang, L. Qin, and B. Hu, “Cold-attack: Jailbreaking llms with stealthiness and controllability,” 2024. [Online]. Available: <https://arxiv.org/abs/2402.08679>

## A Prompts used During Thesis

**Synthetic Data Generation Prompt:** The prompt given below was used for synthetic data generation for fine-tuning of the LLMs during this thesis. The variable context is provided as chunks of the API specification file and the variable examples is substituted by JSON object as shown in Listing 1.

```
Generate data simulating interactions with the NEF API endpoints using
the provided context. Generate at least 20 unique questions.
None of the questions must be repeated, they all must be unique.

context: {context}

For each simulated interaction, the output must be JSON objects
having the following format:

    request: A question about utilizing the NEF API to perform a
specific action.
    api_call: The full URL (taken from the 'path' field) of the
API endpoint being invoked.
    description: A brief summary of the purpose of the API
endpoint.
    method: The HTTP method used to call the API endpoint.
    operation: The operationId of the API endpoint.
    parameters: Any parameters required by the API endpoint, a
comma separated dictionary of key-value pairs.

Example of such data is: {examples}

Your output MUST strictly follow these rules and outline:

- The request, api_call, description, method, operation, and
parameters fields must not contain be empty strings.
- Generated data must not be fake or hallucinated

Generate diverse data covering various endpoints, methods, and
parameter combinations available in the NEF API schema file.
Don't generate fake questions. All of the generated data must be
taken from the provided NEF API schema file.
```

**Listing 7:** Prompt for Data Generation

**Data Scaling Prompt:** The prompt given below was used for scaling the initially generated data by the teacher model, i.e., GPT-4. Here, the variable `json_object` is given an originally generated object per iteration and the complete document of the API specification is provided as context.

```
Understand the given JSON object and generate the request parameter
in 100 different ways. All of them must be unique and not
redundant.
```

```
    {json_object}
```

```
    You may also use the document provided as context to
    understand more. Feel free to rephrase the request parameter.
```

```
    {context}
```

```
    The format of output must be an array of 100 values in the
    following format:
```

```
    [request1, request2, ..., request100]
```

**Listing 8:** Prompt for Data Scaling

**Prediction Generation Prompt for Phi-2:** The following prompt was used to generate prediction data for the evaluation of Phi-2 base model as well as its fine-tuned version. Here again, context is provided dynamically from the input document and question refers to the original query from the user about NEF APIs.

```
Instruct:You are an assistant that answers questions related to NEF
  API.
Return your response as JSON with format provided by the human.
</s>User:Generate a response in JSON format for the given question.
Ensure the JSON object includes the keys 'api_call', 'description', '
  method', 'operation', and 'parameters' with appropriate values.

Question: {question}

{context}

Example format of the expected answer must be a JSON object having the
  following keys:
{{
  api_call: The full URL (taken from the 'path' field) of the API
  endpoint being invoked.
  description: A brief summary of the purpose of the API endpoint.
  method: The HTTP method used to call the API endpoint.
  operation: The operationId of the API endpoint.
  parameters: Any parameters required by the API endpoint, a comma
  separated dictionary of key-value pairs.
}}
Dont make up answers, say i dont know. Put quotes around the key-value
  pairs. \nOutput:
```

**Listing 9:** Prompt to Generate Prediction Data from Phi-2