**LLM-Driven Synthetic Data Generation for API Testing using Natural Language and Data Schema**

DISSERTATION

Submitted in partial fulfillment of the requirements of the

Degree: MTech in AIML

By

**RAGURAMAN T**

2023AA05528

Under the supervision of

**RAGHU RAJENDRAN**

Principal Engineer

Optum Global Solutions

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

Pilani (Rajasthan) INDIA

JUL 2025

# Acknowledgement

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI**

**CERTIFICATE**

This is to certify that the Dissertation entitled **LLM-Driven Synthetic Data Generation for API Testing using Natural Language and Data Schema** and submitted by Mr. **T RAGURAMAN** ID No. **2023AA05528** in partial fulfillment of the requirements of AIMLCZG628T Dissertation, embodies the work done by him under my supervision.

Signature of the Supervisor

Name: **RAGHU RAJENDRAN**

Place: **Chennai**

Designation: **Principal Engineer**

Date: **\_\_-08-2025**

# Abstract

Software testing is an important phase in the software development lifecycle. It ensures that the application we build achieves the purpose we intended to fulfill. With the emergence of Dev-Ops concepts and modern development methodologies, the software development lifecycle has fairly shortened in duration. Applications are built in iterations with ever varying requirements.

One of the biggest problems the developers and tester would face during their testing is the test data creation. The software needs to be tested for various scenarios and edge cases, which requires large volume of test data covering all possible scenarios. In traditional software testing, the test data is manually created, and it suffers from the following issues- time consuming and error prone due to lack of understanding of the data structure and the schema. Testers would need to spend time to understand the data schema co-relate it with the test scenario and then decide what fields are needed and what not. The quality of test data is also another concern as creating a test data which does not conforms to the schema or standards, puts the entire process at risk. In some cases, the production data is used to create the test data by data obfuscation process which has potential to leak confidential data into test environments.

The difficulties faced by the testers in test data creation are numerous and sometimes frustrating. What if there is a tool which can understand plain English and generates the test data in seconds? A tool which can understands the context, in our case the schema of the data, and generates the data based on users’ natural language input as prompts. A tool which can integrate with various other project management and SDLC tools and aids in simplifying the process of test data generation right from the plan/design phase.

Here I intend to create a tool which can be used to generate test data based on a fixed schema. A schema will be passed to the tool so set the context of the test data structure. An LLM based model will be used to understand the requirements for the test data creation passed by the user. The parsed requirement is then used to generate the test data. There are multiple approaches to generate the data, of which I intend to explore 2 approaches and chose the best to implement to achieve our goal. 1. Leveraging the pre-trained LLM to create a model by prompting, 2. Using the NLU concept of LLM to parse and understand the requirement and then generating the data based on a custom logic. Both the approaches have its own pros and cons which will be analyzed, weighed and picked for implementation.

# List of Abbreviations

API Application Programming Interface

CPU Central Procesing Unit

EDI Electronic Data Interchange format

GPU Graphic Processing Unit

JSON Javascript Object Notation

LLM Large Language Model

LoRA Low Rank Adaptation

PEFT Paramtere Efficient Fine tuning

QLoRA Quantized LoRA training

VRAM Video Random Access Memory

# List of Tables

[Table 1 Comparison of Current Data Generation Approaches 12](#_Toc202136641)

# List of Figures

[Figure 1 Low Rank Matrices 16](#_Toc205846385)

[Figure 2 High Level Architecture of the Solution 17](#_Toc205846386)

[Figure 3 Performance comparison of Mistral-7B vs others 19](#_Toc205846387)

[Figure 4 Mistral-7B-Instruct benchmark 19](#_Toc205846388)

[Figure 5 Dataset pattern 20](#_Toc205846389)

[Figure 6 Experiment tracking 22](#_Toc205846390)

[Figure 7 GPU configuration 22](#_Toc205846391)

[Figure 8 Text Generation WebUI - Model and LoRA Adapter loading 24](#_Toc205846392)

# Table of Contents

[Acknowledgement 2](#_Toc205844334)

[Abstract 4](#_Toc205844335)

[List of Abbreviations 5](#_Toc205844336)

[List of Tables 6](#_Toc205844337)

[List of Figures 7](#_Toc205844338)

[Table of Contents 8](#_Toc205844339)

[Chapter 1: Introduction 10](#_Toc205844340)

[Background 10](#_Toc205844341)

[Problem 10](#_Toc205844342)

[Objectives 10](#_Toc205844343)

[Objectives Met 10](#_Toc205844344)

[Chapter 2: Research 11](#_Toc205844345)

[Current Approaches on Test Data Creation 11](#_Toc205844346)

[1. Manual Test Data Creation 11](#_Toc205844347)

[2. Production Data Masking/Obfuscation 11](#_Toc205844348)

[3. Script-Based Data Generation 11](#_Toc205844349)

[4. Record and Replay 12](#_Toc205844350)

[5. Data Generation Tools 12](#_Toc205844351)

[6. AI/ML Based Approaches 12](#_Toc205844352)

[Gaps in Current Methods 12](#_Toc205844353)

[Missing Balance 13](#_Toc205844354)

[A Balanced Approach 13](#_Toc205844355)

[Research on Training methods 13](#_Toc205844356)

[Low Rank Adaptation 14](#_Toc205844357)

[Chapter 3: Literature Review 15](#_Toc205844358)

[Structured Data Generation 15](#_Toc205844359)

[Natural Language to JSON 15](#_Toc205844360)

[Review of LoRA 15](#_Toc205844361)

[Benefits 16](#_Toc205844362)

[Chapter 4: Design & Architecture 17](#_Toc205844363)

[Requirements 17](#_Toc205844364)

[Functional 17](#_Toc205844365)

[Non-Functional 17](#_Toc205844366)

[High-level Design 17](#_Toc205844367)

[Data 17](#_Toc205844368)

[Model (LLM) 18](#_Toc205844369)

[API 18](#_Toc205844370)

[Interfaces 18](#_Toc205844371)

[Scope of the project 18](#_Toc205844372)

[Chapter 5: Technical Implementation 19](#_Toc205844373)

[Model 19](#_Toc205844374)

[Other models considered 20](#_Toc205844375)

[Dataset format and Preparation 20](#_Toc205844376)

[Context In the training data 20](#_Toc205844377)

[Domain Specific Data 20](#_Toc205844378)

[Preparation 21](#_Toc205844379)

[Training 21](#_Toc205844380)

[Framework 21](#_Toc205844381)

[Approach 21](#_Toc205844382)

[Experiment Tracking 21](#_Toc205844383)

[Dataset 22](#_Toc205844384)

[Training Environment 22](#_Toc205844385)

[Training Configurations 22](#_Toc205844386)

[Evaluation 22](#_Toc205844387)

[1. JSON Parse Accuracy 22](#_Toc205844388)

[2. Schema Compliance Score 23](#_Toc205844389)

[3. Field Level Accuracy (Exact Match) 23](#_Toc205844390)

[4. Manual Evaluation 23](#_Toc205844391)

[Inferencing 23](#_Toc205844392)

[Chapter 6: Result Analysis 24](#_Toc205844393)

[Conclusion 25](#_Toc205844394)

[References 26](#_Toc205844395)

# 

# Chapter 1: Introduction

## Background

Software testing plays a critical role in ensuring that applications function as intended. With the rise of DevOps and agile methodologies, development cycles have become shorter and more iterative. One of the major challenges in this fast-paced environment is generating high-quality, schema-compliant test data efficiently. Manual test data creation is time-consuming, error-prone, and often relies on testers’ deep understanding of complex data structures. This has led to a need for intelligent tools that can automate test data generation using natural language inputs and schema context.

## Problem

Manual test data creation is time-consuming, error-prone, and requires deep understanding of complex data schemas. Existing methods struggle to generate diverse, schema-compliant data quickly, especially in fast-paced DevOps environments. There is a need for an intelligent tool that can generate test data from natural language input using predefined schemas.

## Objectives

Objectives of this project are

1. Creating a tool which generates test data
2. Tool allows us to set the context of the test data by passing a schema document
3. Tool allows to customize the test data using user prompts

## Objectives Met

The following tasks haven been completed so far.

1. Research of the current methods in test data generation is done
2. High level design is created
3. Technical implementation choices are made
4. Model selection is completed
5. Training dataset format is decided.
6. Initial implementation of the training is done.
7. Training data preparation- In-progress

# Chapter 2: Research

## Current Approaches on Test Data Creation

I did a thorough research on how the test data is generated in the industry at present to gain an in depth understanding of the test data generation methods and their strengths and weaknesses.

### Manual Test Data Creation

Testers create test data by manually entering values that fulfill specific test case requirements. This often requires a strong understanding of the data schema, business logic, and test objectives.

#### ****Challenges:****

* Time-consuming and labor-intensive.
* Error-prone, especially with complex or large datasets.
* Low scalability for large or frequently updated applications.
* Testers must have deep domain and schema knowledge.
* Difficult to maintain consistency and coverage across tests.

### Production Data Masking/Obfuscation

In this method, real-world data from production systems is copied and then sanitized or anonymized before being used in test environments. The goal is to retain the realism and complexity of actual data—like realistic formats, patterns, and relationships—while removing or altering sensitive and personally identifiable information (PII) to avoid data privacy violations.

#### ***Challenges***:

* Data privacy and security risks, especially if masking is incomplete.
* Compliance issues with GDPR, HIPAA, or other data protection regulations.
* High complexity in correctly masking sensitive fields without corrupting data relations.
* Not ideal for all testing scenarios, especially edge cases or error conditions.

### Script-Based Data Generation

Script-based test data generation involves writing custom code (usually in scripting or programming languages like SQL, Python, Java, or Shell) to create and insert test data into a database or data storage system. These scripts are crafted based on the application’s data schema, business logic, and test case requirements.

The approach allows testers or developers to control the kind of data generated, ensuring it meets specific needs such as format, constraints, and relationships among entities.

#### Challenges:

* Requires skilled developers or testers to write and maintain scripts.
* Scripts may break if the underlying schema changes.
* Limited flexibility without frequent updates.
* Difficult to handle dynamic or highly variable test requirements.

### Record and Replay

Record and Replay is a test data reuse strategy where actual data (or requests/responses) from previous test runs or production-like environments is captured ("recorded") and then used again ("replayed") in subsequent test cycles. This method is commonly used in integration testing, API testing, UI automation, and performance testing scenarios. This method is sometimes referred as A/B testing in the industry.

Rather than generating synthetic data each time, this approach uses real interaction data that has been known to work (or fail), helping verify consistent behaviour over time.

#### Challenges:

* Does not cover new or corner-case scenarios.
* Risk of data staleness as the application evolves.
* Limited flexibility to adapt or scale data to different test scenarios.
* Possibility of repeating hidden defects due to unchanged data.

### Data Generation Tools

Data generation tools are specialized software applications designed to automatically create synthetic test data based on predefined rules, schema definitions, and business constraints. These tools help QA teams, developers, and data engineers generate large volumes of realistic, valid, and varied data that closely mimics real-world datasets—without using actual production data.

#### Challenges:

* May require learning curve and licensing costs.
* Generated data can lack realism unless properly configured.
* Some tools have limited support for complex schema relationships.
* Integration issues with custom or legacy systems.

### AI/ML Based Approaches

Uses machine learning models or large language models (LLMs) to learn patterns from existing data or user instructions, then generates realistic or context-aware test data.

#### Challenges:

* Still an emerging field, with ongoing research and limited tools.
* Generated data may lack control and precision for specific test cases.
* Requires fine-tuning models and handling large compute resources.
* Explainability and traceability of generated data can be limited.
* Risk of bias or security issues if training data isn't well curated.

## Gaps in Current Methods

The above section detailed the various testing and data generation techniques exist in the current industry. They all have their own advantages and disadvantages. The below table shows each methods and their trade-offs.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Approach | Time Consumption | Flexibility | Skillset needed | Privacy |
| Manual | High | Good | High | Poor |
| Masking | Low | Poor | Low | Good |
| Tool Based | Medium | Good | High | Good |
| Record and Replay | Low | Poor | Low | Poor |
| Script based | High | Good | High | Moderate |

Table 1 Comparison of Current Data Generation Approaches

A problem addressed in one technique is not covered in the other. And most of them had the below common issues that needs attention

* Time consuming
* Need of deeper understanding of the data or generation method
* Privacy and compliance
* Lack of flexibility

### Missing Balance

While each of the methods of data generation or preparation addresses a different issue discussed above, there is a no methodology that addresses all of them. For example,

Masking or Obfuscation based method reduces time by sourcing the data from the production but it lacks flexibility in covering edge cases as most majority of production data falls into a straight forward case.

Data Generation Tools may be designed to cover the edge cases but they lacks realism and needs a testers to be technically strong to customize, update and maintain the tool.

Manual generation of data does not need technical understanding but a deeper understanding of data formats

There is clearly a lack of balance between these approaches and there is a need to develop new solutions that addresses all these short comings.

## A Balanced Approach

The previous section emphasized on the missing balance between various data preparation approaches. There is a clear need for newer solutions that try to bridge the gap in terms of balancing the issues each of the approaches suffering from.

The solution should mainly address the followings

**Time Consumption**: Time taken by a tester or developer to generate/prepare data should be less without compromising the quality

**Flexibility**: The solution is flexible enough to generate data with all possible cases and edge cases

**Knowledge of data**: The solution should take over the knowledge of data structures and generation techniques requiring the testers to possess lesser knowledge on data structure and nuances.

**Security**: The solution should synthesize the data and not use the actual data

## Research on Training methods

When we want a LLM to generate new data in the format we want, then the options available to us are

1. Create a LLM specific for the use case
2. Fine tune a pre-trained LLM

Creating an LLM from the scratch is too expensive and we will have to need a large data to train the model not only on the custom data we want, but also on general data to instill the reasoning and generalizing capabilities.

Fine tuning is technique where we take a pre-trained model and further train it on the data we want. Unfortunately this too is expensive, as we want to update all of the weights in the model. Let suppose we choose a 7 billion parameter model, the fine tuning will update all the 7 billion parameters when training with the custom data. The hardware and the GPU required for this task is also too high.

### Low Rank Adaptation

Low Rank Adaptation or LoRA, is a technique which reduces the no of parameter updates in fine tuning. Instead of updating all of the parameters, in LoRA we freeze the weights of the LLM and add layers of trainable low rank matrices. The training process will update this smaller training parameters and thus reducing the need for huge hardware and GPUs.

# Chapter 3: Literature Review

## Structured Data Generation

Generating a structured data like JSON is a bit challenging as the model should follow a specific syntax and semantics. Generating the JSON data without any proper training like in a zero-shot setup has poor results as per this paper [1] on JSON response formatting with LLMs. The paper introduces **StructuredRAG**, a benchmark comprising **six tasks** designed to evaluate how well LLMs can generate **structured outputs** (e.g., JSON) in **zero-shot settings** critical for building compound AI systems that rely on structured responses.

The paper also talks about two prompting strategies, Follow the Format (FF) and f-string prompting which will be tried out in this project.

## Natural Language to JSON

The paper **“LLMs on the Fly: Text-to-JSON for Custom API Calling” [2] introduces a pipeline for converting user-written natural language queries about IoT devices into structured JSON API calls, enabling a virtual assistant to query and visualize data through the FlyThings® API.**

**This paper suggests the following setup:**

* Template-based JSON generation – Randomly create JSON objects following a predefined schema.
* Data augmentation – Use an LLM (Mixtral‑8x7B‑Instruct) to generate diverse query variants for each JSON example through few-shot prompting.
* Supervised fine-tuning – Train the LLM to map natural-language queries to the correct JSON structure, improving precision over pure in-context learning.
* Inference optimization – Deploy the fine-tuned model with quantization (AWQ) and efficient serving (vLLM) on GPU hardware to support real-time performance.

**We will consider the Supervised fine-tuning suggested in this paper to implement in our usecase.**

## Review of LoRA

LoRA is a fine tuning technique introduced in a paper [2] on Jun 2021 to reduce the training effort required in full fine tuning of a pre-trained LLM. A full fine tuning requires all of the models parameter to the retrained which is time consuming and hardware intensive. In this paper a new technique is introduced which freezes the pretrained model weights and injects trainable rank decomposition matrices into each layer of the Transformer architecture. This reduces the number if trainable parameter greatly to the downstream tasks.

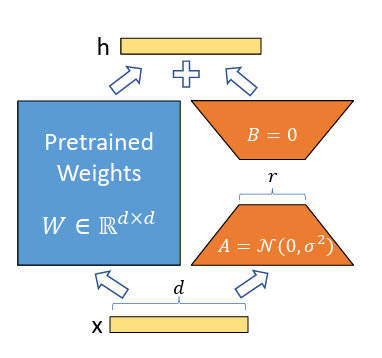


Figure 1 Low Rank Matrices

We start with a pre-trained weight matrix W0∈Rd×k and update it using a low-rank decomposition: , where , and . During training, W0 is kept fixed and only A and B are trainable. The output is computed as:

Both and BA are applied to the same input x, and their results are added together

### Benefits

1. The significant benefit is in the form of reduction in memory and storage usage.
2. No additional inference latency.
3. Multiple layers can be trained and included in the model

# Chapter 4: Design & Architecture

## Requirements

This section will describe the requirements of the solution considering the scope of the project and real world use case.

### Functional

The system should have the following functional requirements

1. Solution should be in the form of a chat interface but should be extensible to other forms via APIs.
2. Solution should allow the user to describe the data required in natural language
3. Solution should accept a schema of the data and generate the data conforming to the schema structure
4. Solution should be capable of generating domain specific data
5. The data generated should be a valid format, e.g., JSON, XML, etc.

### Non-Functional

Following non-functional requirements should be met.

1. Time taken to generate the data should be under 10 seconds.
2. System should be available for 99% of the time.

## High-level Design

The figure below shows a high-level design of the solution.

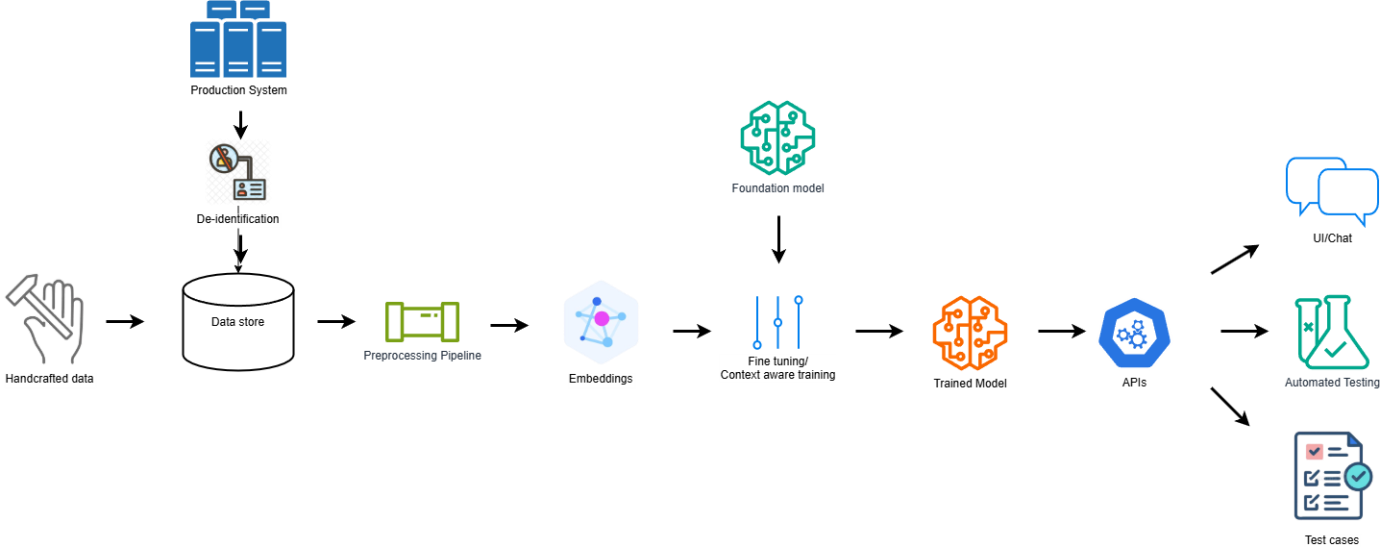


Figure 2 High Level Architecture of the Solution

### Data

The training data for this design can be prepared in two ways

1. Sourced directly from the production systems after de-identifying the user information if any.
2. Hand crafted data covering all possible scenarios

While production data can give various real world scenarios, hand crafted data can provide edge cases and balance of data classes. The solution will support both.

### Model (LLM)

LLM will be used for understanding the user input and reasoning. It will also be fine-tuned with LoRA technique to enable it generate domain specific content. The model will be chosen considering its capability to generate structured data.

### API

The trained model will be served through REST APIs. APIs are the current industry standards when it comes to providing a service. With micro-service architecture, and various options to implement, APIs will be created to expose the model which will be consumed by the end interfaces.

### Interfaces

Once the model is available as an API, the options for interface are endless. The most obvious and meaningful choice will be a chat interface where a user can input the requirement and get the model generate the data.

## Scope of the project

The above design is a aimed at solving the problem with many features included, the project implementation will be limited to proof of concept. For example, the data will only be handcrafted and no production data will be used in this project. The final interface will be chat based UI and no integration to other products will be implemented even though the solution will support it.

# Chapter 5: Technical Implementation

This chapter covers the technical choices and the detailed implementation of the solution.

## Model

Selection of a model is subjective to individual who evaluates the various models and the constraints one has on the requirements. We can basically choose any 7B+ model to do our task but keeping the resources needed to train and the cost in mind, I decided to go with a 7B model which gives good reasoning and multi turn chats.

Mistral-7B-Instruct-v0.3

Mistral 7B Instruct is the instruction tuned version of the base model Mistral 7B from Mistral AI. It is an open source LLM with 7.3 billion parameters and 8K context length. It boasts strong reasoning capabilities and better performance on chat and coding tasks.

Our project which is primarily a chat based solution will benefit from the instruction tuned version of the Mistral 7b model. We also need strong reasoning capability in order to understand the user requirement and generate the data. The below performance chart from the official Mistral site shows the models performance against various other models.

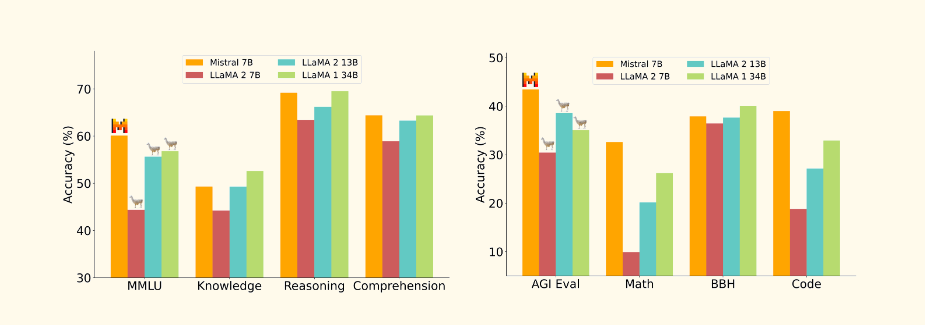


Figure 3 Performance comparison of Mistral-7B vs others

The instruction tuned version of the mistral 7B has outperformed the other models as per the official site.

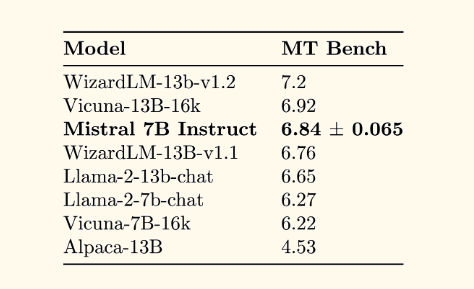


Figure 4 Mistral-7B-Instruct benchmark

### Other models considered

LLaMa 3 8B is a model from Meta and it too has better performance equivalent to the Mistral but the lack of official instruction variant has pushed to choose the Mistral-7B-Instruct.

StarCoder, a model better at handling the coding task is also considered. To keep the model size smaller and training shorter, I have still chosen the Mistral-7B-Instruct.

## Dataset format and Preparation

The data-set, mostly will be handcrafter for this proof of concept. A real world solution will have much better ways to source data from the production systems. The format of the training data will be in instruction-response pairs. In addition to the instruction-response, the context, the schema of the data structure will also be included in the dataset to make the model understand the context and generate the data conforming to the structure in the context.

Mistral suggests the data to be prepared in jsonl format, multi turn instruction-response pairs. The context can be included in the initial instruction. The following figure shows the format of the data as suggested by Mistral.

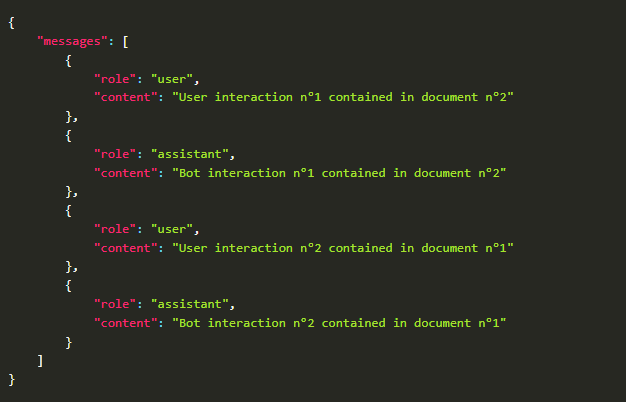


Figure 5 Dataset pattern

### Context In the training data

Context is very important for our use case as the json generated by the model should conform to the schema passed as context. The data in json will then be transformed to the chat template shown below

<s>[INST] Context: json schema goes here.\nUser: Hello, generate a sample data for the given schema [/INST] expected json data goes here</s>

The above data is then tokenized and processed further to train the model.

### Domain Specific Data

One of the main reasons for this fine tuning is to make the model capable of generating domain specific content. Mistral 7B has good json generation capability but the custom data formats needs to fine tuned into it.

For example, the healthcare field has various EDI formats, which are special kinds of CSV-like data formats with their own grammar.

ISA\*00\* \*00\* \*01\*SENDERID \*01\*RECEIVERID \*141016\*2359\*U\*00401\*000000001\*0\*P\*:~  
GS\*PO\*SENDERID\*RECEIVERID\*141016\*2359\*000000001\*X\*004010~  
ST\*850\*0001~  
BEG\*00\*SA\*PO12345\*20141016~  
N1\*ST\*SHIP TO NAME~  
N1\*BY\*BILL TO NAME~  
IT1\*1\*2\*EA\*10.00\*\*VP\*PRODUCT123~  
IT1\*2\*3\*EA\*20.00\*\*VP\*PRODUCT456~  
CTT\*2~  
SE\*10\*0001~  
GE\*1\*000000001~  
IEA\*1\*000000001~

Current LLM models are not well trained with this formats and training any further new formats has to be done on top of the base model in the form of fine-tuning.

### Preparation

Training data is created using a custom python code using faker library. Faker is a python library to generate fake data for all kinds of software testing. Faker is used in a custom python code to generate training data needed for the fine tuning.

## Training

### Framework

**Huggingface:** The finetuning of our model is carried out using Huggingface training framework. Huggingface is an ecosystem which provides training framework with set of tools, libraries and APIs to train and fine-tune machine learning models especially transformers—without having to write everything from scratch.

It also provides various foundation models including the Mistral 7B for further training and fine tuning with LoRA.

### Approach

**PEFT:** PEFT (Parameter-Efficient Fine-Tuning) is a library designed to adapt large pre-trained models to different downstream tasks without updating all of their parameters, which would otherwise be prohibitively expensive. Instead, PEFT techniques fine-tune only a small set of additional parameters, drastically reducing computational and storage requirements while achieving performance close to that of full fine-tuning. This approach makes training and storing large language models (LLMs) far more feasible on consumer-grade hardware. Huggingface provides framework to perform PEFT based fine tuning which is used in the project to reduce the size of the hardware needed.

**QLoRA:** Quantized Low-Rank Adaptation, is an advanced parameter-efficient fine-tuning method designed for large language models (LLMs). It builds upon the foundation of LoRA (Low-Rank Adaptation) by adding quantization to dramatically reduce memory usage. This implementation have used *bitsandbytes* library to quantize the model before tuning.

### Experiment Tracking

**MLFlow: MLflow** is an **open-source platform** for managing the end-to-end machine learning lifecycle*.* This project uses MLflow to track the training paramteres, metrics and artifacts. Huggingface training framework has built-in support to MLFlow which makes logging metrics and tracking experiment easier. In addition to the metrics published by the framework, few additional custom metrics are also published specific for the project use case.

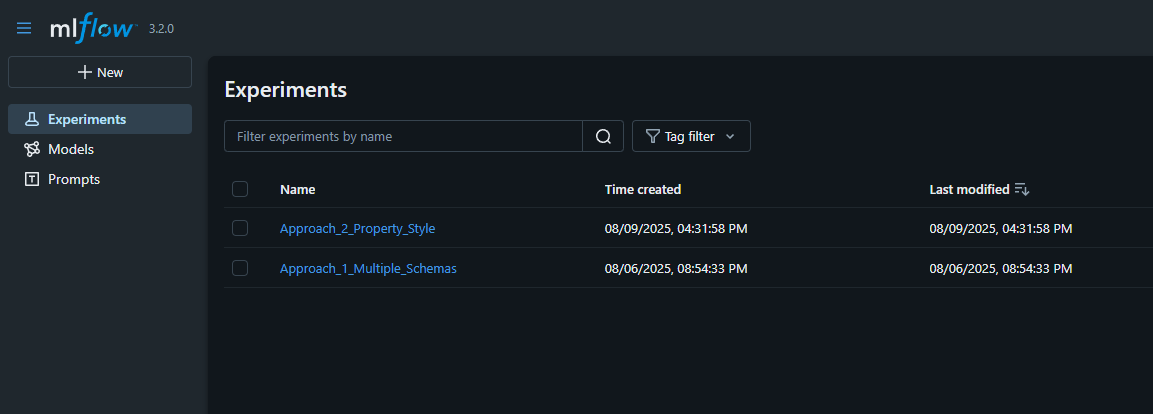


Figure 6 Experiment tracking

### Dataset

The training data is split into 3 sets as training 80%, validation 20% and testing 10%. Validation set is used for evaluation at the end of each epoch. Test set is used on the final model to test the performance of the model.

This project used 2 variants of training data

**Full Schema Style** – training samples contains full JSON schema in the context with corresponding JSON response.

**Property Style** – training samples will contain only the property schema and its corresponding gold response.

### Training Environment

Training is carried out in consumer desktop with 32GB RASM and GPU VRAM of 16GB. The GPU model used is nvidia RTX 5070TI which has 8960 cuda cores and is sufficient to perform quantized fine tuning on mid-range models like 7-8B

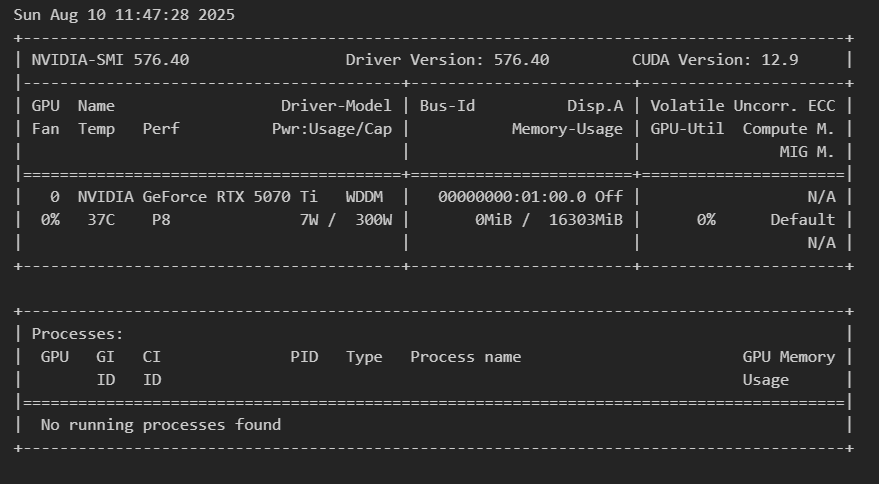


Figure 7 GPU configuration

### Training Configurations

The below configurations were used during the training

1. 4 bit quantization (QLoRA)
2. Batch size : 1 (to fit the training in the available VRAM)
3. Gradient Accumulation – 8 (this will simulate a batch size of 8)
4. Learning rate – 0.0002
5. No of Epochs : 4-6
6. Checkpoints at the end of each epoch
7. Loss metric : Cross Entropy
8. Evaluation at the end of each epoch
9. Gradient checkpointing enabled (this will reduce the memory requirement by not storing all of the activation in the forward pass and recalculates it on backward pass)
10. Best model is stored at the end of training based on evaluation loss

### Metrics

By default huggingface frameworks captures and logs a set of metrics into the MLFlow. It includes evaluation loss, training loss, learning rate, runtime, etc. These metrics are captured for each experiment and each training run performed.



Figure 8 Evaluation loss

The above graph tracks the evaluation loss on the validation dataset. As per the graph, the loss is lesser in the penultimate epoch at which point afterwards the models started to overfit.



Figure 9 Training loss



Figure 10 Learning rate

Each new training iteration starts with a learning rate of .0002 and automatically adjusted by the training framework.

### Model

Final model is selected based on the evaluation and validation against the test dataset. Huggingface training is performed with model saved at the end of each epoch and validation happens against the validation set. The evaluation loss metric is logged and at the end of the training, huggingface framework will save the model which has least evaluation loss.

In addition to that, a custom evaluation is performed against the test set to calculate custom metrics listed in next section. This metric will also be referred and the final model is chosen based on the custom metrics.

## Evaluation

Evaluating the fine-tuned LLM's ability to generate syntactically and semantically valid JSON data requires a multifaceted approach. The evaluation process in this project focuses on the structural correctness of the generated output, adherence to the given schema, and field-level semantic accuracy. While this evaluation focuses on JSON data, the method is similar for other data formats. The following metrics and methods were used:

### JSON Parse Accuracy

This metric evaluates whether the generated output is syntactically valid JSON. It is a binary measure where a prediction is marked valid if it can be parsed without errors.

**Method:** Using a JSON parser to attempt parsing the model output.

**Formula:**

### Schema Compliance Score

This metric evaluates whether the generated JSON conforms to the expected schema passed as context.

**Method:** Validate each output against a JSON schema using the jsonschema library.

**Formula:**

### Field Level Accuracy (Exact Match)

This checks if individual fields in the generated output match the reference output for a given test input.

**Method:** Compare values of each field in generated JSON with reference JSONs.

**Formula:**

Field level accuracy may be an important metric but in our project, the model needs to manifest a good level of generalization and should be able to generate different sample data and not always generate the same data. So this metric will only considered to a level that only the fields with fixed values are evaluated.

### Manual Evaluation

A manual review was optionally conducted for a small sample of generated outputs to assess:

**Correctness:** Does the output logically fit the prompt?

**Completeness:** Are all relevant fields populated?

**Realism:** Does the data resemble production-like examples?

## Inferencing

This project uses the “Text Generation WebUI” to serve the model through chat interface. Text Generation WebUI is an **open-source, browser-based interface** for running and chatting with large language models. It makes it easy to load, run, and interact with LLMs without writing Python code. It supports multiple models and can run on a smaller GPU/CPUs.

Text Generation WebUI supports loading and switching between LoRA adapters and this makes it easy to serve the fine-tuned models without the hassle of building and maintaining inferencing code.

# 

Figure 11 Text Generation WebUI - Model and LoRA Adapter loading

# Chapter 6: Result Analysis

# Conclusion

Pending

# References

[1] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen - LoRA: Low-Rank Adaptation of Large Language Models, Jun 2021 - https://arxiv.org/abs/2106.09685

[2] Miguel Escarda-Fernández, IñigoLópez-Riobóo-Botana, SantiagoBarro-Tojeiro, LaraPadrón-Cousillas, SoniaGonzalez-Vázquez, Antonio Carreiro-Alonso and Pablo Gómez-Area - LLMs on the Fly:Text-to-JSON for Custom API Calling - https://ceur-ws.org/Vol-3729/d3\_rev.pdf

[3] Connor Shorten, Charles Pierse, Thomas Benjamin Smith, Erika Cardenas, Akanksha Sharma, John Trengrove, Bob van Luijt - StructuredRAG: JSON Response Formatting with Large Language Models - https://arxiv.org/abs/2408.11061