FPT Education
FPT UNIVERSITY

MINISTRY OF EDUCATION AND TRAINING

FPT UNIVERSITY

AI Capstone Project Document

Agent Programmatic Integration Testing

Group	
Group Members	Tran Bac Nam - Leader - SE171773
	Tran Trung Hieu - Member - SE173132
	Tran Quoc Truong - Member - SE173295
Supervisor	Huynh Van Thong - Ph.D.
Capstone Project code	AIP491-FA25AI20-GFA25AI09

HCMC, Dec 2025

Contents

Listings

List of Figures

List of Tables

1 Introduction

1.1 Background

In the context of modern software systems increasingly relying on microservices and API-based communication, ensuring the stability, performance, and security of APIs has become a critical requirement. Recent studies show that APIs already account for more than 80% of global internet traffic [1, 2], highlighting the importance of robust testing strategies. However, manual testing methods are often time-consuming, prone to missing edge cases, and unable to keep up with the rapid pace of CI/CD development cycles [3]. This situation creates a pressing need for automated API testing solutions that are both scalable and cost-efficient. The emergence of Large Language Models (LLMs) provides new opportunities: automatically analyzing API specifications, generating valid test cases, executing them, and evaluating results in a more intelligent and efficient manner compared to traditional approaches [4, 5].

1.2 Objectives

The objective of the *Agent Programmatic Integration Testing (APIT)* project is to build an automated API testing tool that leverages LLMs to minimize manual intervention while optimizing testing costs. Specifically, the system is expected to: (i) analyze standardized API specifications (OpenAPI, Swagger), (ii) generate valid test cases and prioritize those with higher value, (iii) automatically execute and classify results into pass/fail/exception, and (iv) produce structured reports through an interactive dashboard that supports real-time monitoring.

1.3 Problem Statement

Currently, many organizations still rely on manual testing or traditional automation tools that require complex configurations and lack flexibility. This results in high costs, incomplete coverage, and difficulties in adapting to frequent API changes. Moreover, test case generation is often manual and dependent on tester expertise, which can easily overlook risky scenarios. The problem, therefore, is to design an intelligent automated testing solution that can learn from API documentation to generate test cases and evaluate results, ensuring API quality with optimized cost and resources [6].

1.4 Scope and Limitations

The project focuses on developing an AI-driven automated API testing system, including the ability to analyze and process standardized API documents, generate and manage RESTful test cases, execute tests, and report results via a web-based dashboard. Out of scope are mobile application development, UI or advanced performance testing, support for languages other than English and Vietnamese, and offline deployment. Limitations of the project include the lack of real-world testing data, risks arising from unstandardized API documentation, and the potential high cost of operating LLMs when applied to large and complex API systems.

1.5 Project Positioning

The Agent Programmatic Integration Testing (APIT) project is strategically located within the standard Software Development Life Cycle (SDLC), concentrating its efforts on the crucial **Testing and Integration** phase. As depicted in Figure 1, APIT does not replace the overall development process but serves as an intelligent, automated enhancement for Quality Assurance (QA). The entire process consists of six main phases:

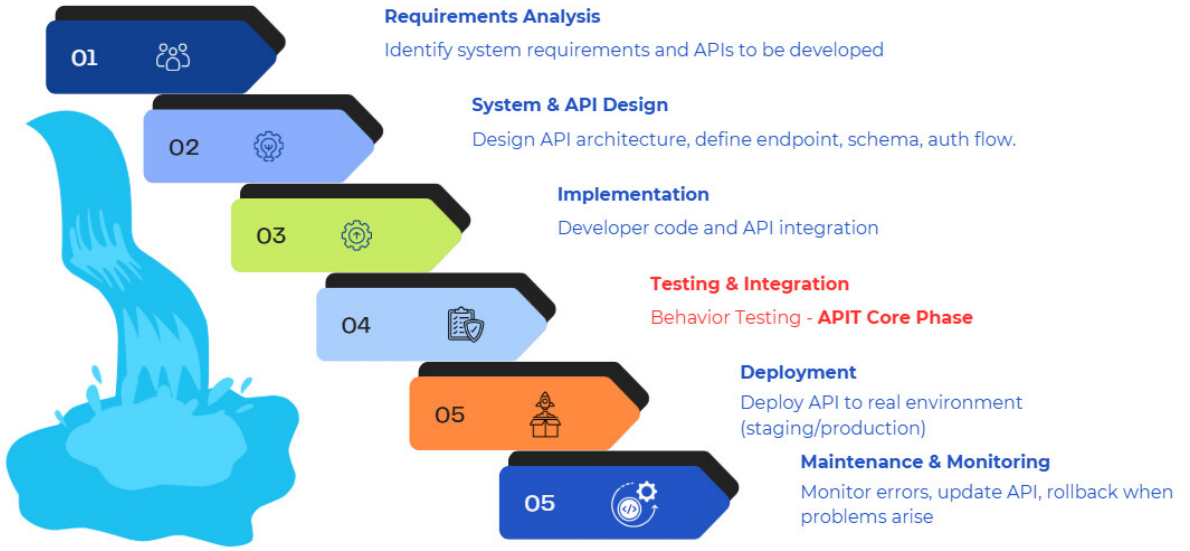


Figure 1: Software Development Life Cycle (SDLC) Model and APIT Project's Positioning in the Testing Phase.

1. **Requirements Analysis:** Identify system requirements and APIs to be developed.
2. **System & API Design:** Design API architecture, define endpoint, schema, and authentication flow.
3. **Implementation:** Developer code and API integration.
4. **Testing & Integration (APIT Core Phase):** This is the primary focus of the APIT project, where automated behavior testing is performed using LLMs.
5. **Deployment:** Deploy API to the real environment (staging/production).
6. **Maintenance & Monitoring:** Monitor errors, update APIs, and perform rollbacks when problems arise.

By centralizing its function in Phase 4, the APIT system ensures that critical testing and validation steps are executed early and automatically, directly addressing the pain points of high costs and incomplete coverage mentioned in the Problem Statement.

2 Project Management Plan

2.1 Team Structure and Roles

There are 3 members in our group, their roles and responsibilities are shown in table 1

Table 1: Team Roles

Name	Role	Overall Tasks
Tran Bac Nam	Leader	- Control and monitor the progress of the project. - Maintain communication between members. - Propose a model for further training/fine-tuning.
Tran Trung Hieu	Data Analyst	- Collect and analyze data for the project. - Process and synthesize data based on the existed ones.
Tran Quoc Truong	System Architecture Designer	- Design the system required for the project. - Optimize the system performance.

2.2 Communication Plan

In the course of 15 weeks, our plan for communication are as follows:

1. Progress Tracking
   (a) Method: Direct Messages
   (b) Frequency: Bi-Daily
   (c) Responsible: Leader
2. Decision Making Meeting
   (a) Method: Direct meetings, online meetings via Discord or Google Meet
   (b) Frequency: Weekly
   (c) Responsible: Leader

3 Literature Review

3.1 Traditional Approaches

API testing has long been conducted either manually or through rule-based automation. Manual testing, while simple to implement, is inherently time-consuming, error-prone, and unsuitable for large-scale or frequently changing APIs [3]. Automation frameworks such as JUnit, TestNG, or Selenium extensions provided partial support but were never designed specifically for APIs, leading to limitations in scalability and maintainability.

3.2 Established Tools and Frameworks

To address these challenges, a range of specialized API testing tools have emerged. Postman has become the industry standard due to its intuitive interface and collection-based execution, yet it still relies heavily on manually defined test cases [7]. Apache JMeter is widely used for load and performance testing, though its functional testing capabilities require significant configuration effort [8]. RestAssured and PyTest, as code-centric frameworks, offer high flexibility but demand substantial programming expertise. Newman extends Postman's automation features but inherits the same dependency on human-defined scripts. Overall, these tools are effective in execution and reporting but fall short in intelligent test generation.

3.3 AI-driven Approaches

Recent advances in Artificial Intelligence (AI), particularly Large Language Models (LLMs), have introduced new opportunities in software testing. Researchers have demonstrated the use of natural language processing techniques to interpret API documentation and generate test scenarios automatically. For example, *KAT* applies LLMs with dependency graphs to generate dependency-aware test cases [4], while *DeepREST* leverages reinforcement learning to uncover hidden constraints in REST APIs and improve coverage [6]. Similarly, *APITestGenie* explores generative AI to produce executable test scripts directly from OpenAPI specifications [5]. These works highlight the potential of LLMs to optimize test selection by prioritizing endpoints with higher risks, improving both cost-efficiency and coverage. However, most current AI-driven methods remain experimental, often limited to proof-of-concept studies without full integration into CI/CD pipelines.

3.4 Comparative Summary

Table 2 summarizes the strengths and limitations of key API testing approaches. It highlights the gap between traditional execution-focused tools and emerging AI-driven solutions that promise more intelligent test generation and optimization.

Table 2: Comparison of API testing tools and approaches

Tool/Approach	**Strengths**	**Limitations**
Postman	User-friendly, widely adopted, supports automated collections	Relies on manual test design, limited intelligence
JMeter	Strong for load and performance testing	Functional testing requires heavy setup, less intuitive
RestAssured / PyTest	Flexible, integrates well into codebases	Requires programming skills, steep learning curve
Newman	Automates Postman collections, CI/CD integration	Dependent on predefined test cases, no test generation
AI-driven (LLMs)	Potential for automatic test generation, intelligent prioritization	Mostly experimental, limited large-scale deployment

3.5 Research Gap

Although mature tools exist for execution and reporting, and AI-driven methods show promise in test generation, there remains a lack of an integrated solution that combines intelligent test case creation, automated execution, and cost-aware reporting. This gap provides the motivation for the *APIT* project, which aims to leverage LLMs to deliver an end-to-end, scalable, and resource-optimized API testing system.

4 Methodology

4.1 Data Collection

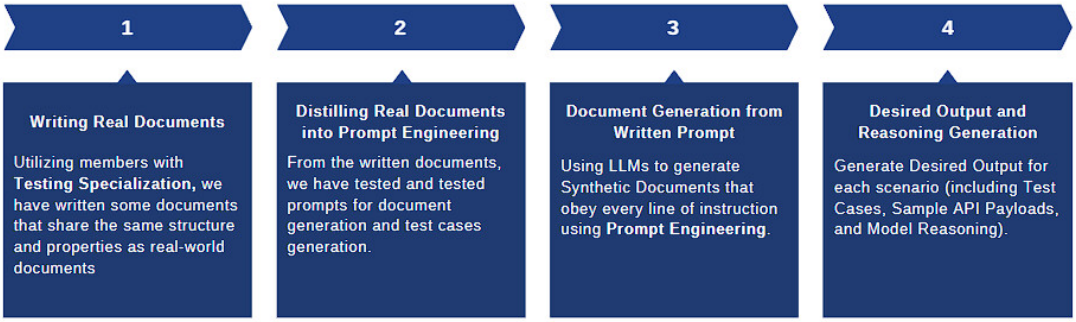


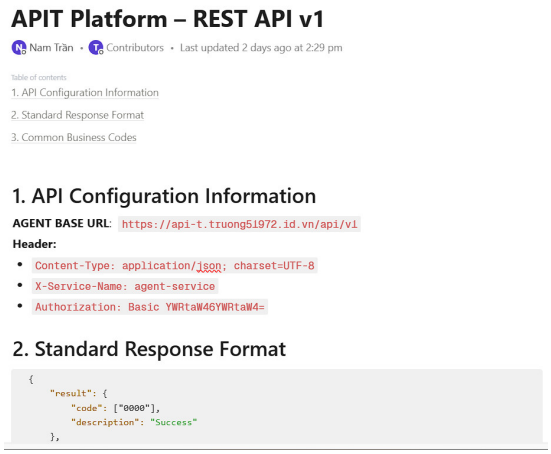
Figure 2: Data Collection Pipeline

Due to the lack of data availability and the private nature of Real-world API Documents, we have decided to synthesize our data for fine-tuning models. According to Nadas et al., 2025 [9], data synthesis can be used for low-resource situations while having to control the unpredictability of Large Language Models (LLMs) for the best result. Yu et al., 2023 [10] also proposed the Topic/Controlled Generation method, which can help us control the output of the LLMs to be as close and diverse as the original documents.

For these reasons above, we have created a data collection pipeline (as in fig. 2). This pipeline consists of 4 steps: **Writing Real Documents ⟶ Distilling Real Documents into Prompt Engineering ⟶ Document Generation from Written Prompt ⟶ Desired Output and Reasoning Generation.**

4.1.1 Writing Real Documents

Our member, Truong, a specialized tester working for FPT Software, has written some API documents that mimic all possible properties of a real document.


APIT Platform – REST API v1

Nam Trần • Contributors • Last updated 2 days ago at 2:29 pm

Table of contents

1. API Configuration Information
2. Standard Response Format
3. Common Business Codes

1. API Configuration Information

AGENT BASE URL: https://api-t.truong51972.id.vn/api/v1

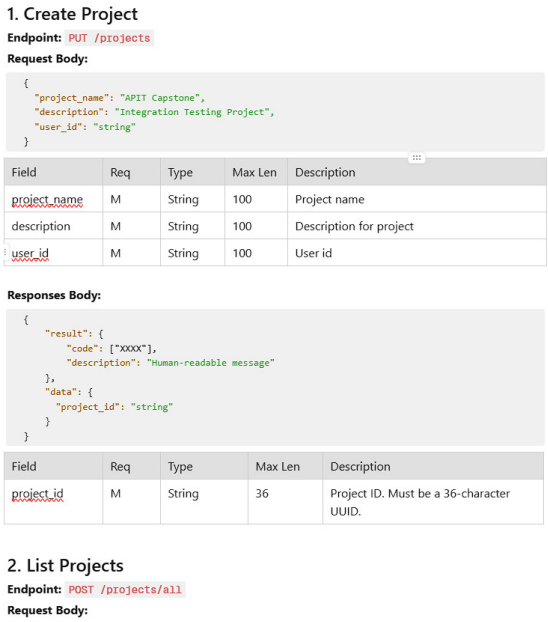
Header:

- Content-Type: application/json; charset=UTF-8
- X-Service-Name: agent-service
- Authorization: Basic YWRtaW46YWRtaW4=

2. Standard Response Format

```
{
    "result": {
        "code": ["0000"],
        "description": "Success"
    },
```

Figure 3: API Configuration Document


1. Create Project

Endpoint: PUT /projects

Request Body:

```
{
  "project_name": "APIT Capstone",
  "description": "Integration Testing Project",
  "user_id": "string"
}
```

Field	Req	Type	Max Len	Description
project_name	M	String	100	Project name
description	M	String	100	Description for project
user_id	M	String	100	User id

Responses Body:

```
{
    "result": {
        "code": ["XXXX"],
        "description": "Human-readable message"
    },
    "data": {
      "project_id": "string"
    }
}
```

Field	Req	Type	Max Len	Description
project_id	M	String	36	Project ID. Must be a 36-character UUID.

2. List Projects

Endpoint: POST /projects/all

Request Body:

Figure 4: API Detail Document

4.1.2 Distilling Real Documents into Prompt Engineering

From the written documents mentioned in section 4.1.1, we have extracted all the most important properties of a real document. Then we created a prompt that can generate a distilled version of the written documents. See Prompt 2.

Prompt Description

- Lines 1-16: Configure the domain, API functionality and the structure of the output document.
- Lines 20-35: Provide an overall instruction to LLMs.
- Lines 37-178: Give the strict output structure as well as instructions for each section.

To see the full version of the prompt, see Prompt 2.

4.1.3 Document Generation from Written Prompt

After creating and testing prompt 2 in section section 4.1.2.
Firstly, we controlled the output of the LLM by passing our chosen **Domain, API functionality, Documentation Structure,...** to the Configuration part in the prompt 2. (To further see our chosen configurations, see section 4.2.)
Then, we push the configured prompt to the LLM (here we used Gemini-2.5-Pro for cheaper cost while maintaining output quality). Finally, we have to iterate through every output document to filter out unqualified output documents.

4.1.4 Desired Output and Reasoning Generation

After generating qualified documents in section 4.1.3, we then created a prompt purely to generate test cases from the given document.

Prompt Description

- Lines 1-8: Give general instructions to the LLM as well as define the input structure.
- Lines 12-52: Give further instructions about the details of test cases and Chain of Thought (CoT).
- Lines 54-104: Define the output structure.
- Lines 106-129: Define field descriptions and special keywords with explanations.
- Lines 133-136: Control input document and tell the model to return an output.

To see the full version of the prompt, see Prompt 3.

4.2 Data Analysis

In other to control our desired output, we have analyzed and controlled the configuration for data generation. In other to achieve that goal, we used Topic/Controlled Generation used by Yu et al., 2023 [10].

4.2.1 Train - Val - Test Split

our dataset consists of 1258 samples. Our data split for Train - Val - Test will be approximately 8:1:1.

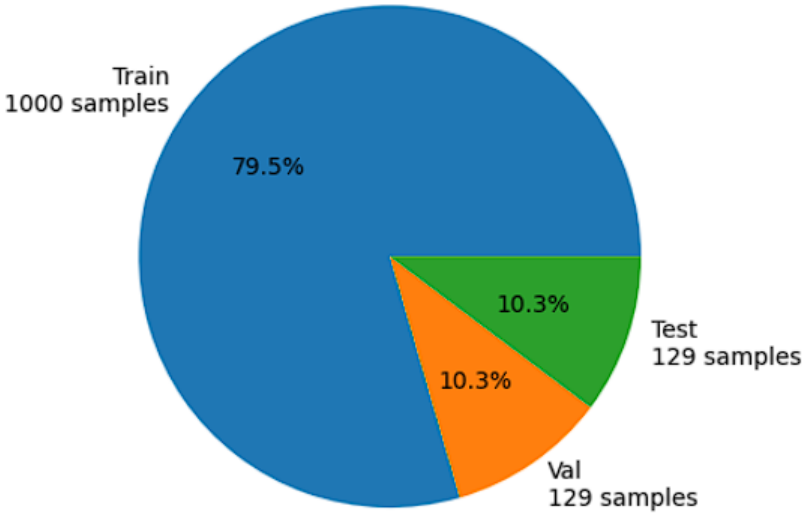


Figure 5: Train - Val - Test Split Ratio

4.2.2 Language

Our goal was to have a model that supports 2 languages: **English and Vietnamese**. We aimed for a 1:1 ratio for each language. After generation, we filtered the synthesized data and achieved roughly the same ratio as our goal.

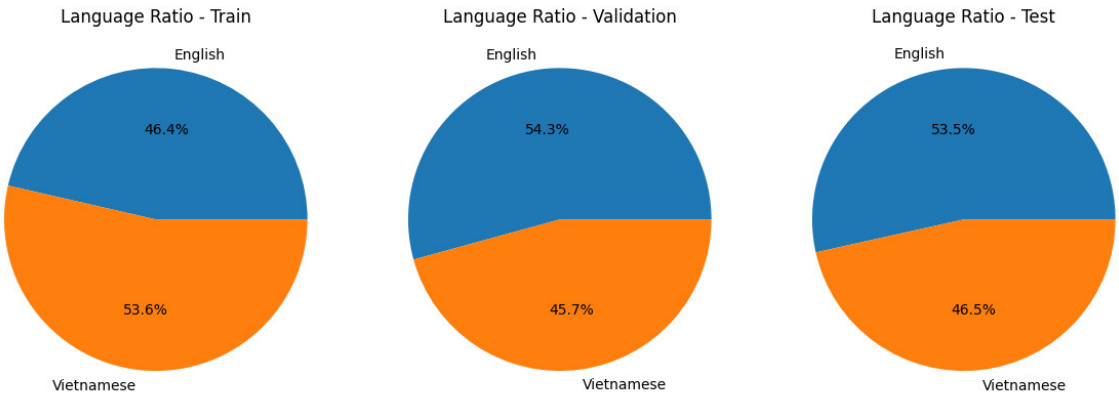


Figure 6: Data Language Ratio

4.2.3 Structure

We provided a structure controlling for **API Documentation, API Detail, Behaviour Rules and Data Test,** and wanted to have an as equally distributed data structure as possible. As a result, we obtained our data with a distribution that is approximately the same as a uniform distribution.

- **API Documentation and API Detail:** plain text, table, list.
- **Behaviour Rules and Data Test:** table, list.

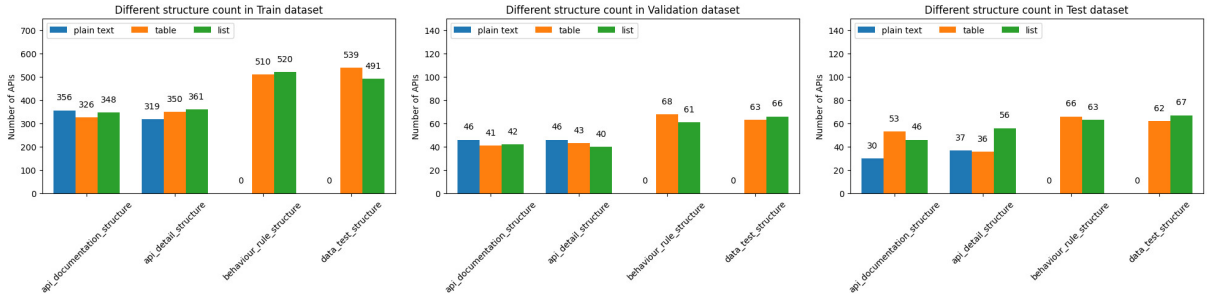


Figure 7: Attributes' Structure Type Distribution

4.2.4 Attribute Count

We provided an attribute count control for **API Documentation, Request Body Field, Behaviour Rules and Data Test,** and aimed to have an as equally distributed data structure as possible. After synthesizing, we achieved a distribution that matched what we wanted.

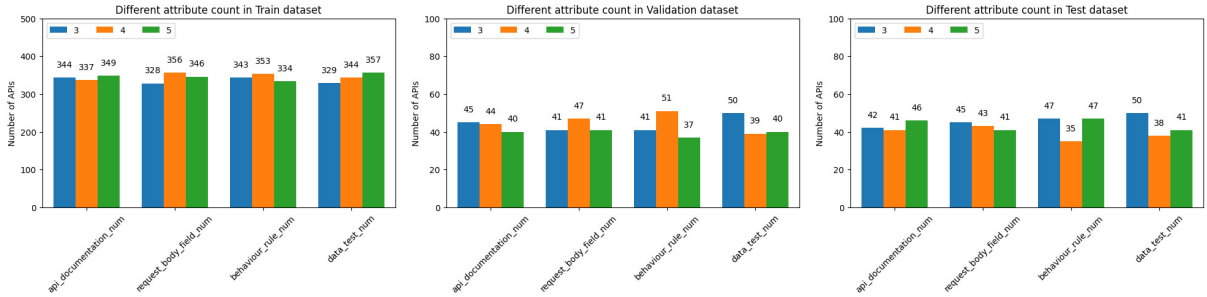


Figure 8: Attribute Count Distribution

4.2.5 Domain and API Functionality

We would like to focus on domains that serve in Banking and Sales because they are the most widely used domains in web apps. For the distribution, Sales should be more dominant than Banking (Sales:Banking ratio is roughly 8:2).

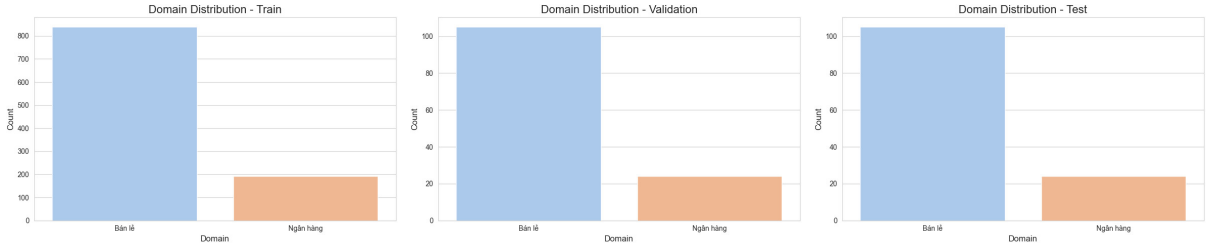


Figure 9: Domain Distribution

For API Functionalities, Payment will be the dominant functionality while maintaining a proper distribution for each functionality.

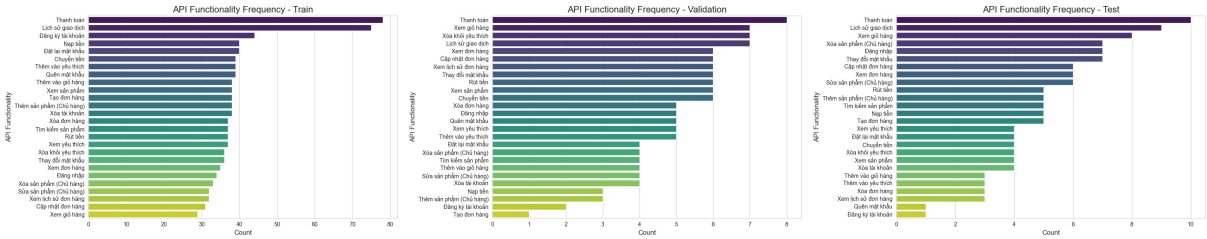


Figure 10: API Functionality Distribution

4.3 Training Methodology

To efficiently fine-tune large language models while minimizing computational resources, we adopt Parameter-Efficient Fine-Tuning (PEFT) techniques, specifically Low-Rank Adaptation (LoRA) and its memory-optimized variant, Quantized Low-Rank Adaptation (QLoRA). Both approaches significantly reduce the number of trainable parameters compared to full fine-tuning while achieving comparable performance.

4.4 Parameter-Efficient Fine-Tuning with Adapters

One of the most widely used approaches in Parameter-Efficient Fine-Tuning (PEFT) is the insertion of small, trainable **adapter modules** into a frozen pre-trained Transformer model. Unlike full fine-tuning, which updates all parameters of the model (often billions in size), adapter-based methods keep the original pre-trained weights completely frozen and only train the newly added adapter parameters. This drastically reduces memory consumption, training time, and the risk of catastrophic forgetting.

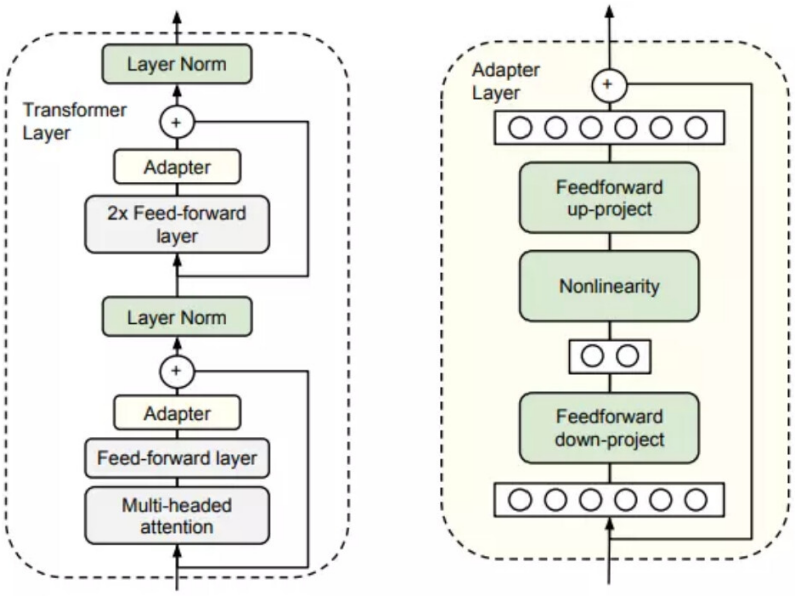


Figure 11: Structure of a Transformer layer augmented with bottleneck adapters. **Left**: Overall view of a Transformer block with two adapters inserted in parallel after the multi-head attention and feed-forward sublayers, respectively. Residual connections and layer normalization are preserved. **Right**: Detailed internal architecture of a single adapter module consisting of a down-projection, non-linear activation, up-projection, and a residual connection.

4.4.1 Transformer Layer with Adapters (Left Diagram)

1. The input first passes through the standard **multi-head self-attention** sublayer.
2. An **adapter module** is inserted **in parallel** immediately after the attention sublayer: the original activation is fed into the adapter, and the adapter output is added back to the original activation via a residual connection.
3. Layer normalization is applied (placement can vary, but is commonly used before or after the addition).
4. The result then flows into the **feed-forward** sublayer (typically two linear layers with a non-linearity in between).
5. A second **adapter module** is placed after the feed-forward sublayer in the same parallel/residual manner.
6. Finally, the output of the entire Transformer layer is produced.

This parallel insertion ensures that at initialization (when adapter weights yield zero or identity output), the model behaves exactly like the original pre-trained model.

4.4.2 Internal Structure of an Adapter Module (Right Diagram)

Each adapter is a classic bottleneck architecture designed to keep the number of trainable parameters extremely low:

1. **Feedforward down-projection**: A linear layer reduces the hidden dimension from the model's dimension d_{model} (e.g., 4096 or 5120) to a much smaller bottleneck dimension d_{bot} (typically 8–256).
2. **Non-linearity**: A non-linear activation function (ReLU, GELU, or SwiGLU) is applied in the low-dimensional space.
3. **Feedforward up-projection**: A second linear layer expands the representation back to the original model dimension d_{model}.
4. **Residual connection**: The output of the up-projection is added to the adapter's input, and this sum becomes the adapter's final output.

This original adapter design [11, 12] laid the foundation for many subsequent PEFT methods (including Houlsby adapters, Pfeiffer adapters, and later LoRA/QLoRA variants). In the following sections, we describe how we build upon this concept using the more memory-efficient Low-Rank Adaptation (LoRA) and Quantized LoRA (QLoRA) techniques, which can be viewed as special cases or improvements of the general adapter paradigm.

4.4.3 LoRA: Low-Rank Adaptation

Instead of updating the full pre-trained weight matrix $W_0 \in \mathbb{R}^{d\times k}$, LoRA injects trainable low-rank decomposition matrices into selected layers. The weight update ΔW is factorized as the product of two low-rank matrices:

$$W = W_0 + \Delta W = W_0 + BA, \tag{1}$$

where $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$, and the rank $r \ll \min(d, k)$ (typically 8–64). During training:

- W_0 remains frozen,
- B is initialized to zero,
- A is initialized with random Gaussian values.

At initialization, $\Delta W = BA = 0$, ensuring no deviation from the original model behavior. Only A and B are updated during fine-tuning, reducing both memory footprint and training cost.

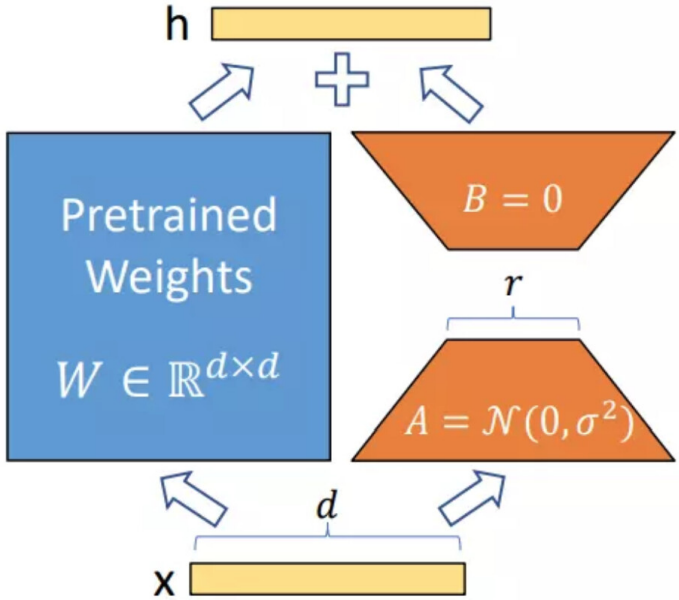


Figure 12: Low-rank decomposition in LoRA. The weight update $\Delta W = BA$ is represented by two small matrices of rank r. Only these matrices are trained.

4.4.4 QLoRA: Quantized Low-Rank Adaptation

QLoRA extends LoRA by combining three key memory-saving techniques:

1. 4-bit NormalFloat (NF4) quantization of the base model weights,
2. Low-rank adapters (identical to LoRA),
3. Double quantization and paged optimizers to further reduce memory overhead.

This combination enables fine-tuning of 33B–70B models on a single 24GB GPU and 7B–13B models on consumer-grade hardware with as little as 6–8GB VRAM.

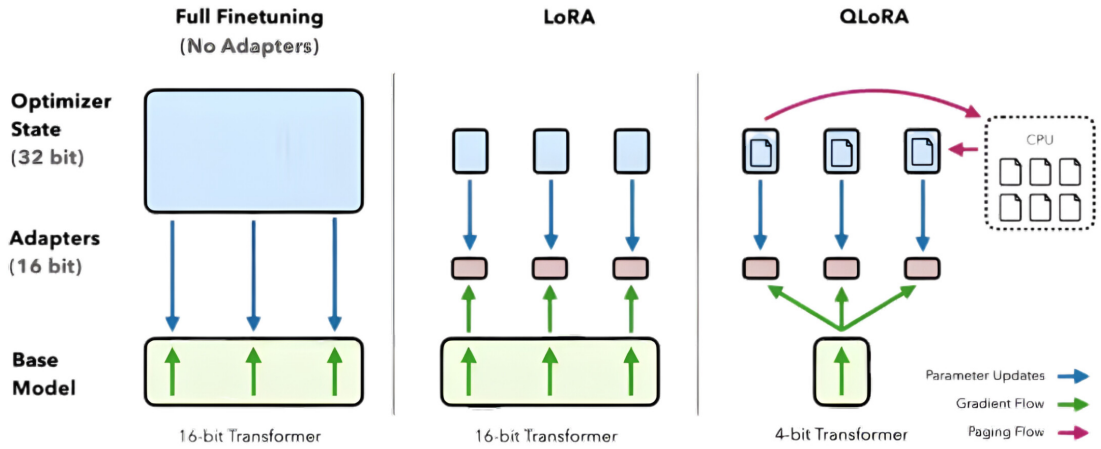


Figure 13: Comparison of VRAM usage and training time across full fine-tuning, LoRA, and QLoRA (with Unsloth). QLoRA reduces memory requirements dramatically, enabling 8B model fine-tuning in under 10 minutes on a single GPU.

4.4.5 Implementation with Unsloth

We perform all experiments using the Unsloth framework [13], which provides highly optimized implementations of LoRA and QLoRA. Key advantages include:

- 2–5× faster training compared to standard Hugging Face PEFT,
- Native support for 4-bit and bfloat16 training,
- Built-in gradient checkpointing, paged attention, and optimized kernels,
- Seamless compatibility with modern open-source models (Llama-3, Mistral, Gemma-2, Phi-3, Qwen-2, etc.).

With Unsloth + QLoRA, we successfully fine-tune 8B-class models in under 10 minutes on a single A100 40GB GPU — a substantial improvement over traditional approaches that require hours or multiple high-end GPUs.

4.5 Training Process

4.5.1 Training Strategy

Instead of teaching the model entirely new factual knowledge, our core objective is to make it **adopt the correct format, structure, and reasoning style** required in the Automated Programming Interface Testing (APIT) domain. This includes strict adherence to formal conventions such as:

- Functional Requirements (FR) must always be phrased as “The system shall ⟨…⟩”.
- Every test case must explicitly include an Expected Result field.
- Test steps must be written in imperative, sequential language with precise preconditions and postconditions.
- Non-functional requirements (NFR) follow standardized templates (performance, security, scalability, etc.).

By exposing the model to thousands of high-quality, consistently formatted examples during fine-tuning, it gradually internalizes these domain-specific writing and reasoning patterns rather than merely memorizing content. This instruction-following and format-adherence paradigm has proven far more effective and robust than traditional fact-based fine-tuning for specialized documentation tasks.

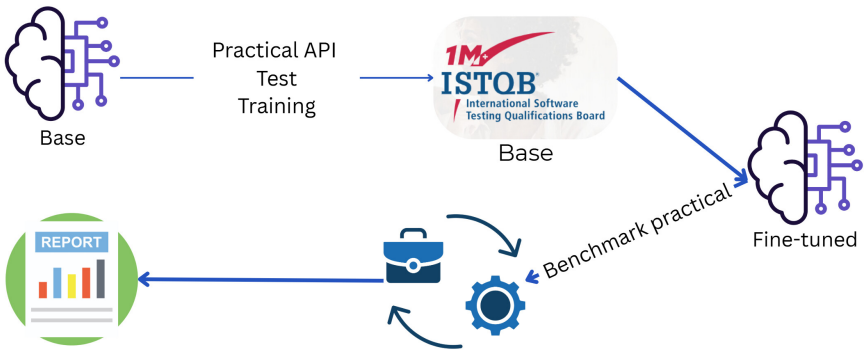


Figure 14: Detailed hyperparameter configuration used for fine-tuning both Qwen-2.5-3B and Llama-3.2-3B models. All parameters are identical across the two base models to ensure fair comparison.

4.5.2 Fine-Tuning Hyperparameters

We fine-tune both Qwen-2.5-3B and Llama-3.2-3B using identical hyperparameter settings (Figure 14) to enable direct performance comparison. Key configurations are summarized below:

- **Base model**: 4-bit quantized (`load_in_4bit=True`) versions of Qwen-2.5-3B and Llama-3.2-3B
- **PEFT method**: LoRA with rank $r = 32$ and scaling factor $\alpha = 32$
- **Target modules**: All linear layers in query, key, value, output projection, gate, up, and down projections (`q, k, v, o, gate, up, down`)
- **Trainable parameters**: 59M (Qwen, Llama) — approximately 0.8–1.1% of total model size
- **Optimizer**: 8-bit paged AdamW (`adamw_8bit`) with learning rate 2×10^{-4}
- **Training regime**: Maximum sequence length of 4096 tokens, per-device batch size of 1, gradient accumulation of 2 steps (effective batch size = 2), maximum 515 training steps
- **Memory optimization**: Unsloth-accelerated gradient checkpointing and 4-bit training
- **Evaluation**: Performed every 30 steps using `eval_strategy = steps`

These settings, combined with Unsloth's optimized kernels [13], enable complete fine-tuning of both 7–8B models on a single 40GB A100 GPU in under 15 minutes while maintaining numerical stability and high throughput.

The consistent hyperparameter choice across Qwen and Llama models ensures that any observed performance differences stem from architectural and pre-training disparities rather than training configuration artifacts.

4.6 Model Selection

We selected **Qwen-2.5-3B** (Alibaba Cloud) and **Llama-3.2-3B** (Meta) as the base models for this project for the following reasons:

Both models are among the strongest openly available 3-billion-parameter LLMs as of late 2025, consistently ranking at the top of major leaderboards (Open LLM Leaderboard, Arena-Hard, MT-Bench) in instruction-following and reasoning capabilities [14, 15, 16].

Crucially, they receive first-class optimization in the Unsloth framework, including custom 4-bit training kernels that deliver 2–5× speedups and approximately 70 % lower VRAM consumption compared to generic implementations. As a result, full LoRA fine-tuning (rank 32) could be completed in only 6–7 hours on free Kaggle/Colab GPUs (T4/P100 with 16 GB), satisfying our strict reproducibility and zero-cost requirements.

Official, high-quality 4-bit quantizations (GGUF, GPTQ, AWQ) are instantly available on the Hugging Face Hub for both models, eliminating quantization-related artifacts and lengthy preparation steps. Additionally, the models differ in tokenizer type (Byte-Level BPE vs. TikToken-style), normalization strategy, and pre-training corpora, providing meaningful architectural and data diversity for fair comparative analysis.

Finally, being released under permissive licenses by two leading institutions ensures long-term availability, active community support, and minimal risk of deprecated checkpoints. Larger models (7B and above) were explicitly excluded because they exceed the memory limits of free-tier GPUs, violating our accessibility goal.

In summary, Qwen-2.5-3B and Llama-3.2-3B represent the optimal combination of raw performance, training efficiency, ecosystem maturity, and practical deployability for a resource-constrained yet high-impact capstone project.

4.7 Training Process

4.7.1 Training Strategy

Our primary goal is not to teach the model new factual knowledge, but to instill the **correct domain-specific writing style, formal structure, and reasoning patterns** required for Automated Programming Interface Testing (APIT) documents. Through repeated exposure to thousands of high-quality, consistently formatted examples, the model learns strict conventions such as:

- Functional requirements must follow the template: "The system shall ⟨action⟩ ⟨object⟩ ⟨condition⟩".
- Every test case must contain a clearly defined **Expected Result**.
- Test steps are written in imperative form with precise preconditions and postconditions.
- Non-functional requirements adhere to standardized performance, security, and scalability templates.

This **instruction-following + format-adherence** approach yields significantly more robust and consistent outputs compared to traditional content-memorization fine-tuning.

4.7.2 Fine-Tuning Configuration

Both Qwen-2.5-3B and Llama-3.2-3B models are fine-tuned using **identical hyperparameters** (Table 3) to ensure a fair and controlled comparison.

Table 3: Hyperparameters used for LoRA fine-tuning (identical for both base models).

Category	**Parameter**	**Value**
Base Model	Model name	Llama-3.2-3B / Qwen-2.5-3B
	Max sequence length	4096
	4-bit quantization	True (`load_in_4bit`)
	Trainable parameters	59M (Qwen, Llama)
LoRA Configuration	PEFT method	LoRA (Low-Rank Adaptation)
	Rank (r)	32
	LoRA alpha	32
	Target modules	q, k, v, o, gate, up, down
	Gradient checkpointing	Unsloth-optimized
Optimizer & Schedule	Optimizer	8-bit paged AdamW
	Learning rate	2×10^{-4}
	Max training steps	515
	Per-device batch size	1
	Gradient accumulation steps	2 (effective batch size = 2)
	Evaluation strategy	Every 30 steps

Table 4: Training hardware and runtime (approximate values — replace with your actual logs).

Resource	**Qwen-2.5-3B**	**Llama-3.2-3B**
GPU type	Kaggle T4 (15 GB)	Kaggle T4 (15 GB)
Peak VRAM usage	9.2–9.8 GB	9.5–10.2 GB
Total training time	6 hours 12 minutes	6 hours 38 minutes
Average step time	42 seconds	46 seconds
Framework	Unsloth [13] + Hugging Face Transformers	

Thanks to Unsloth's highly optimized 4-bit training kernels, gradient checkpointing, and custom Triton kernels [13], both 3B models can be fully fine-tuned on a single consumer-grade GPU in approximately 6–7 hours. All experiments reported in this work were conducted on free GPU instances (P100 / T4) generously provided by the Kaggle notebook platform. This combination of Unsloth's efficiency improvements and Kaggle's accessible high-performance computing resources makes the entire fine-tuning and experimentation cycle extremely cost-effective, reproducible, and accessible to individual researchers without requiring institutional or commercial cloud budgets.

The identical hyperparameter configuration across the two base models (Qwen-2.5-3B and Llama-3.2-3B) ensures that any observed differences in downstream performance are attributable solely to architectural choices and pre-training data rather than training artifacts.

4.8 Evaluation Methods

4.8.1 Testcase Generation System Accuracy Evaluation

We first define the notation used throughout this section:

Symbol	Meaning
i	Index of an API record (test sample)
$\|A\|$	Test case content of input A
$\|B\|$	Test case content of input B
$\mathscr{P}_i$	Set of test cases **predicted** by the model for record i
$\mathscr{G}_i$	Set of **ground-truth** test cases for record i (human-written)
$\|\mathscr{P}_i\|$	Number of predicted test cases for record i
$\|\mathscr{G}_i\|$	Number of ground-truth test cases for record i
TP_i	Number of correctly matched predicted $\leftrightarrow$ GT pairs
FP_i	Number of predicted test cases with no acceptable GT match (extra or wrong)
FN_i	Number of ground-truth test cases with no acceptable predicted match (missed)

Table 5: Notation used for evaluation metrics

Evaluate the quality of the test cases automatically generated by the LLM using the well-established information retrieval metrics: Precision, Recall, and F1 Score. These metrics are computed at the individual test-case level by comparing the model's output against human-written ground-truth (GT) test cases.

Purpose

The goal is threefold:

1. Quantify how many of the generated test cases are actually correct and relevant (Precision).
2. Measure how completely the model covers the test cases that should have been produced (Recall).
3. Provide a single balanced score (F1) that penalises both over-generation and under-generation.

Matching Strategy Between Predicted and Ground-Truth Test Cases

Because test cases are structured JSON objects containing a mix of strings, numbers, booleans, and nested structures, a multi-tier matching strategy is required:

1. **Fuzzy String Matching** (for textual fields such as descriptions, expected results, etc.)

$$\text{Similarity}(A,B) = 1 - \frac{\text{Levenshtein}(A,B)}{\max(|A|,|B|)}$$

where $\text{Levenshtein}(A,B)$ is the standard Levenshtein (edit) distance defined recursively as

$$\text{Lev}(A,B) = \begin{cases} |A| & \text{if } |B| = 0 \\ |B| & \text{if } |A| = 0 \\ \min \begin{cases} \text{Lev}(A_{1..m-1}, B) + 1 \\ \text{Lev}(A, B_{1..n-1}) + 1 \\ \text{Lev}(A_{1..m-1}, B_{1..n-1}) + 1_{(A_m \neq B_n)} \end{cases} & \text{otherwise} \end{cases}$$

A predicted string is considered a match if Similarity$(A, B) \geq 0.85$.

2. **Numeric Matching with Tolerance** (for numeric fields such as prices, quantities, IDs, timestamps, etc.)

$$\text{Relative Error} = \left| \frac{\mathscr{P}_i - \mathscr{G}_i}{\mathscr{G}_i} \right|$$

$$\text{similarity} = \begin{cases} 1 & \text{if } \frac{|\mathscr{P}_i - \mathscr{G}_i|}{|\mathscr{G}_i|} \leq \delta \\ 0 & \text{otherwise} \end{cases}$$

The value matches if Relative Error $\leq \delta$, where $\delta = 0.05$ (5% tolerance). Additionally, an absolute tolerance window is applied when the ground-truth magnitude is very small.

3. **Exact Matching** used for category fields (e.g. HTTP method, status code, boolean flag) and for keys must be identical with the following formula:

$$\text{match} = \begin{cases} 1 & \text{if } \mathscr{P}_i = \mathscr{G}_i \\ 0 & \text{otherwise} \end{cases}$$

A predicted test case is marked as a True Positive (TP) only if all of its fields satisfy at least one of the above criteria with some ground-truth test case (one-to-one matching with maximum bipartite matching to avoid double-counting).

Classification of True Positives, False Positives, and False Negatives

Before matching, we attempt to parse the model's JSON output for each record i. If the output is malformed, empty, contains no valid test cases, or raises any parsing exception, we treat it as a critical failure: $\mathscr{P}_i = \emptyset$. This yields $\text{TP}_i = 0$, $\text{FP}_i = 0$, and $\text{FN}_i = |\mathscr{G}_i|$ — all ground-truth test cases are considered missed.

For successfully parsed outputs, we construct a bipartite graph between predicted test cases $\mathscr{P}_i$ and ground-truth test cases $\mathscr{G}_i$ using the multi-tier similarity score (fuzzy string + numeric tolerance + exact fields). Maximum-weight bipartite matching (Hungarian algorithm) is then applied to obtain a strict one-to-one assignment. The classification of every test case is determined exhaustively by the following decision table:

Table 6: Complete TP/FP/FN classification rules (including parsing failures)

Situation			Resulting label	
Pred. exists?	**GT exists?**	**Matched?**	**Predicted**	**GT**
Yes	Yes	Yes	TP (counted once)	Covered (not FN)
Yes	Yes	No	FP	FN*
Yes	No	—	FP	—
No	Yes	—	—	FN
No	No	—	—	—
Output unparseable / empty / malformed			FP = 0, FN = $\lvert\mathscr{G}_i\rvert$	

*Only if this GT case has no matching prediction elsewhere

Detailed interpretation of each row:

- **Row 1**: Model correctly generates a test case that matches a GT case → contributes one TP.
- **Row 2**: After optimal matching, some predicted cases remain unpaired (hallucinated or insufficiently similar) → FP; some GT cases may also remain unpaired → FN.
- **Row 3**: Model generates a test case with no counterpart in GT → pure FP.
- **Row 4**: Model completely omits a required GT test case → pure FN.
- **Bottom row (parsing failure)**: If the model returns invalid JSON, empty output, or crashes on a record (e.g., 5 out of 130 records are unparseable), we set $\mathscr{P}_i = \emptyset$ → **all ground-truth test cases of these records are counted as FN**, with no FP credited. Example: 130 records → 125 successfully parsed and matched normally, 5 failed → the 5 failed records contribute 0 TP, 0 FP, and $\sum_{failed} |\mathscr{G}_i|$ FNs (typically 20–30 FNs if each has 4–6 GT cases).

This design has two important consequences:

- The one-to-one matching prevents score inflation from duplicated valid test cases.
- Parsing failures are heavily penalised by converting all associated GT cases into pure FNs — reflecting the fact that an unreliable generator (even on a small fraction of inputs) is unacceptable in production settings.

Evaluation Metrics

For each API record i, we compute per-record metrics:

$$\text{Precision}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i}, \quad \text{Recall}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i}, \quad \text{F1}_i = 2 \cdot \frac{\text{Precision}_i \cdot \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}$$

Overall performance across the entire dataset is reported in two standard ways:

- **Macro-averaged**: $\frac{1}{N}\sum_{i=1}^{N} \text{Precision}_i$, $\frac{1}{N}\sum_{i=1}^{N} \text{Recall}_i$, $\frac{1}{N}\sum_{i=1}^{N} \text{F1}_i$
- **Micro-averaged (Overall)**: computed on the summed counts

$$\text{Overall Precision} = \frac{\sum \text{TP}}{\sum \text{TP} + \sum \text{FP}}, \quad \text{Overall Recall} = \frac{\sum \text{TP}}{\sum \text{TP} + \sum \text{FN}}$$

Complete Evaluation Pipeline

1. Load model predictions and ground-truth CSV files.
2. Parse and normalise JSON strings into Python dictionaries.
3. For each record, apply the multi-tier matching algorithm with bipartite matching to assign TP/FP/FN.
4. Compute per-record and global Precision, Recall, and F1 scores.
5. Export detailed per-test-case results and visualise metric distributions.

This rigorous, threshold-based yet tolerant matching strategy strikes a practical balance between strict exact matching (too brittle for real LLM outputs) and overly permissive semantic matching (risk of false acceptance), making it well-suited for evaluating structured test-case generation tasks.

4.8.2 Reasoning Evaluation

For our Reasoning Evaluation, we need a method where we can properly see the **Logic Validity, Coherence and Groundedness** of the generated text.

We did our researches on traditional methods such as BLEU [17] and ROUGE-L [18] and found out that these metrics heavily penalize models using text-matching [19].

After that, we looked for more modernized metrics (for example BERTScore [20] and BARTScore [21]) utilizing Small Language Models to transform texts into vector embeddings and then calculate for similarity. These metrics will meet some problems in niche text generation tasks due to the fact that they cannot detect nuance in different specific scenarios [19].

With the advancement of LLMs, they are more and more utilized in evaluating text models to compensate for the disadvantages of previous metrics [19]. For such reasons, we chose **LLM-as-a-Judge** for our Reasoning Evaluation.

Implementing **LLM-as-a-Judge** does have some challenges, such as potential biases and lack of reliability [22] . These limitations can be mitigated using text pairwise comparison integrated using Prompt Engineering [22].

From what we have said so far in this section, we would like to introduce our Scoring System.

4.7.2.1. Scoring Criteria

We would like to introduce 3 criterias as metrics for our reasoning evaluation:

- **Logic Validity / Correctness:** This measures whether the logical steps in the model's reasoning are sound and free of error, and if the final conclusion is correct.
- **Coherence:** This assesses the readability and organizational flow of the reasoning. It checks if each step clearly and naturally follows from the preceding steps.
- **Groundedness / Factuality:** This measures whether the reasoning is factually accurate and grounded in the original problem statement or context provided.

Table 7: Logic Validity

Score	Description
5	**Perfect.** Every step is logically sound, mathematically correct (if applicable), and the final answer is correct. The reasoning matches the golden reasoning's logical flow exactly or proposes an equally valid, sound, and correct alternative.
4	**Mostly Valid.** The reasoning is logically sound and the final answer is correct, but one or two minor, non-critical logical steps are slightly confusing, redundant, or inefficient compared to the golden reasoning.
3	**Partially Valid.** Contains a **minor logical error** or flaw that, surprisingly, still leads to the correct final answer, OR the reasoning is mostly correct but the **final answer is incorrect** due to a late-stage mistake (e.g., calculation error).
2	**Significantly Flawed.** Contains a **major logical error** early in the process that fundamentally breaks the chain of reasoning, leading to an incorrect final answer.
1	**Completely Invalid.** The reasoning is entirely nonsensical, illogical, or based on incorrect premises, leading to an incorrect final answer.
0	**No Reasoning.** No discernible reasoning trace is provided, or the output is irrelevant.

Table 8: Coherence

Score	Description
5	**Perfect Flow.** The reasoning is exceptionally clear, well-structured, and easy to follow. Each step builds logically and smoothly on the last, resembling the clarity of the golden reasoning.
4	**Coherent.** The flow is good, but a step might jump slightly ahead or rely on an assumption that isn't explicitly stated, making it slightly less optimal than the golden reasoning.
3	**Adequate.** The reasoning is mostly coherent, but the sequence of steps is awkward, or there are noticeable organizational issues that require extra effort to follow.
2	**Poor Flow.** Jumps are frequent, intermediate steps are missing, or the ordering is confusing, significantly disrupting the train of thought.
1	**Disjointed.** The steps appear randomly ordered or are completely disconnected from each other.
0	**No Reasoning.** No discernible reasoning trace is provided.

Table 9: Groundedness

Score	Description
5	**Perfectly Grounded.** All claims, facts, and numbers are correctly derived from or explicitly supported by the initial context/prompt. No factual errors or hallucinations.
4	**Minor Deviation.** The reasoning contains one minor factual error or introduces a trivial, unneeded external fact, but this does not affect the core logical argument.
3	**Partially Grounded.** Contains one non-critical but clear factual error or pulls in external information that is slightly irrelevant to the core problem.
2	**Major Factual Error.** Contains a significant factual error or "hallucination" that compromises the reliability of the entire argument.
1	**Highly Ungrounded.** The reasoning is built primarily on incorrect facts or information that contradicts the provided context.
0	**No Contextual Reliance.** The reasoning shows no attempt to engage with the factual information provided in the prompt.

4.7.2.2. Prompt

For this section, we constructed a prompt for our evaluation. See Prompt 4.

Prompt Description

- Line 1: Provide a brief instruction.
- Lines 3-29: Provide the Scoring System.
- Lines 31-48: Provide the input which consists of the input API Document, Golden Reasoning (acting as ground truth), and the model reasoning (to be evaluated).
- Lines 50-55: Provide a clear instruction that helps LLMs with their judgment.
- Lines 57-77: Provide a required JSON output format for easier result post-processing.

4.7.2.3. Pipeline

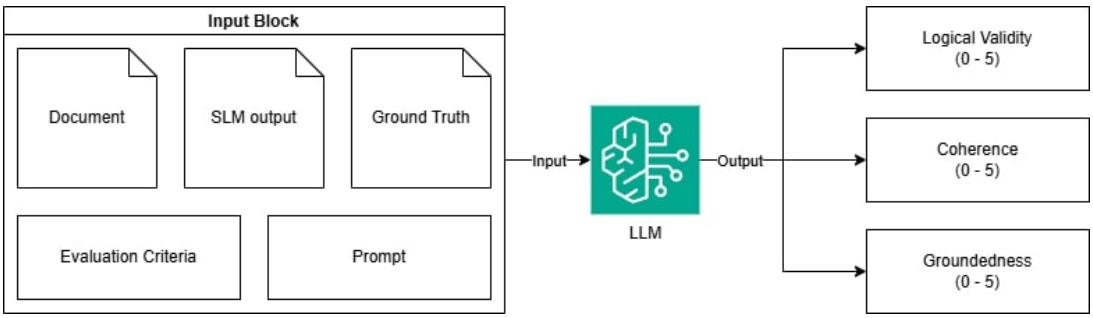


Figure 15: LLM Evaluation Pipeline

Key Components:

- **Input Block:** Gathers all necessary data for evaluation, including the **Document** (context), the **SLM output** (the reasoning generated by the model being evaluated), the **Ground Truth** (the expected reasoning), **Evaluation Criteria** and the **Prompt** used to generate the output.
- **LLM (Evaluator):** A Large Language Model (acting as a judge) processes the combined input.

- **Output Metrics:** The LLM produces scores (on a scale of 0-5) across three distinct quality dimensions:
    1. Logical Validity
    2. Coherence
    3. Groundedness

4.7.2.4. Equations for Scoring

Scoring by each Criterion:

$$S_i(y_{gt}, y) = \frac{1}{N}\sum_{i}^{N}\Phi(y_{gt}, y) \tag{2}$$

Where:

- y_{gt} is the Ground Truth, y is the SLM output.
- N is the number of test samples.
- S_i is the score for each criterion i.
- ϕ is the LLM's Judgment Score. The score ranges from 0 to 5. Higher score means better result.

Weighted Score:

We calculated $WeightedScore$ for the purpose of flattening the result so that we can efficiently compare models.

Since this was meant to be a Reasoning Evaluation task, $W_{LogicalValidity}$ will be the most respected criterion. $W_{Groundedness}$ is more respected than $W_{Coherence}$ because we expect the model to strictly follow the information given in the document.

$$WeightedScore = \sum_{i}(S_i \times W_i) \tag{3}$$

Where:

- S_i (from eq. (2))
- W_i is the weight for each criterion i.
    - $\hookrightarrow$ $W_{LogicalValidity} = 0.5$
    - $\hookrightarrow$ $W_{Coherence} = 0.2$
    - $\hookrightarrow$ $W_{Groundedness} = 0.3$
- $WeightedScore$ ranges from 0 to 5. Higher score means better result.

5 System Design and Implementation

The proposed system is an end-to-end automated test generation and execution framework that transforms unstructured API documentation and existing code repositories into executable test suites with minimal human intervention. The architecture is divided into four sequential stages, as illustrated in fig. 16, fig. 17, fig. 18 and fig. 19.

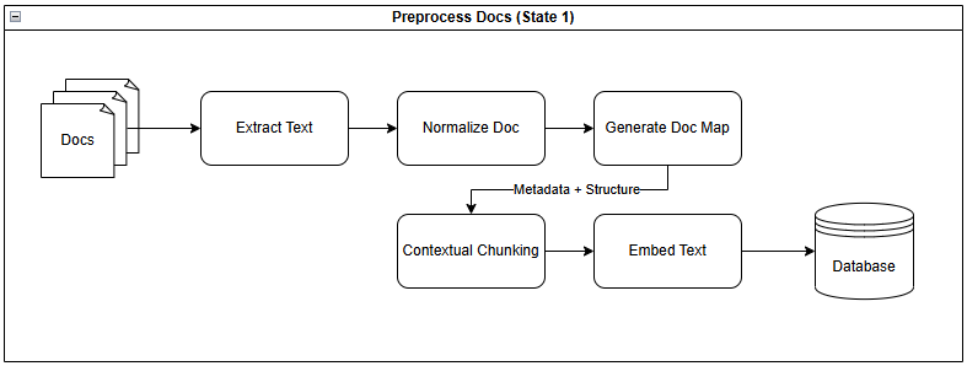


Figure 16: Stage 1 – Preprocess Docs

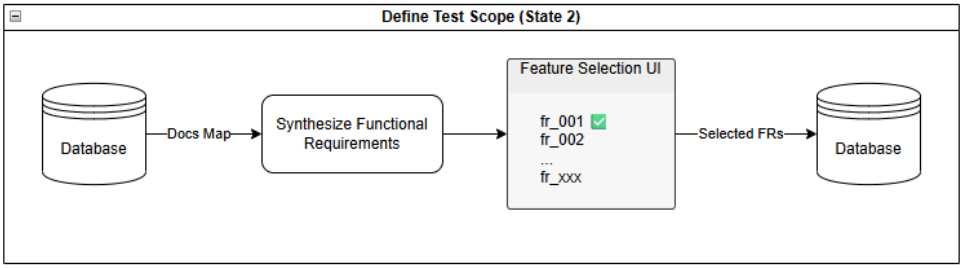


Figure 17: Stage 2 – Define Test Scope

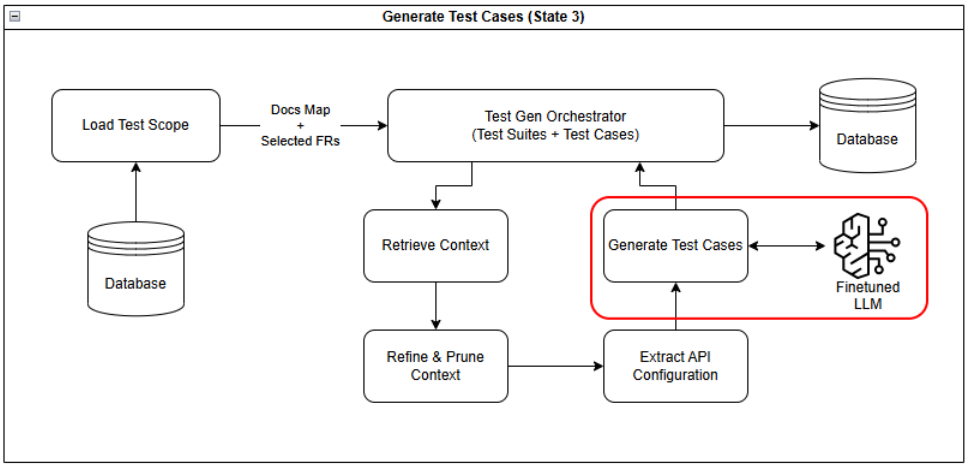


Figure 18: Stage 3 – Generate Test Cases (core LLM stage)

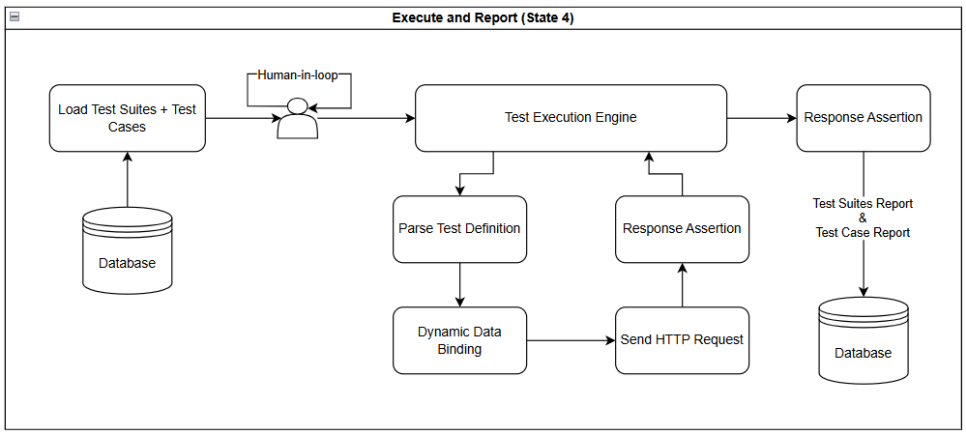


Figure 19: Stage 4 – Execute and Report

Overall system architecture consisting of four sequential stages. Stage 3 (highlighted in red) is the core component that leverages a finetuned large language model combined with retrieval-augmented generation.

5.1 Stage 1: Document Preprocessing

The first stage ingests raw documentation (PDFs, Docs, Drive, etc.) and converts them into a structured, queryable knowledge base.

- **Text Extraction**: Depending on the input format, we employ `pypdf`, `PDFMiner`, or `unstructured.io` to extract raw text while preserving reading order.
- **Document Normalization**: Headers, footers, page numbers, and boilerplate content are removed. Code blocks and tables are identified and separated.
- **Contextual Chunking**: Documents are split into semantically coherent chunks (typically 300–600 tokens) using a sliding-window approach with metadata preservation (section title, API endpoint, HTTP method).
- **Embedding Generation**: Each chunk is embedded and associated metadata are stored in a vector database.
- **Document-to-Requirement Mapping (Doc Map)**: A lightweight index that maps original document sections to functional requirements is persisted for traceability.

5.2 Stage 2: Test Scope Definition

In this human-in-the-loop phase, testers define which functionalities should be covered.

- The previously built *Doc Map* is loaded and presented through a web-based Feature Selection UI.
- Testers select relevant functional requirements (FRs) by checking boxes next to identifiers such as `fr_001`, `fr_002`, etc. (see Figure 17).
- Selected FRs are persisted in the database and serve as the scope for subsequent automated generation.

This step dramatically reduces scope creep and ensures generated tests align with current testing priorities.

5.3 Stage 3: Automated Test Case Generation

This is the core stage where a finetuned large language model generates complete test suites and test cases.

1. **Load Test Scope + Selected FRs** from the database.
2. **Retrieval-Augmented Context Building**:
    - Perform similarity search in the vector store for each selected FR.
    - Retrieve the most relevant chunks.
    - Refine and prune redundant or outdated context using a small LLM critic.
3. **API Configuration Extraction**: Automatically infer base URL, authentication scheme (Bearer, API key, OAuth), and common headers from documentation and existing test code.
4. **Test Generation Orchestrator**:
    - A finetuned Qwen-2.5-3B model receives the curated context, selected FRs, and inferred API configuration.
    - The model first generates a `Test Suite` YAML/JSON skeleton (metadata, setup/teardown).
    - For each FR, it generates multiple `Test Cases` containing description, input data, expected HTTP status, and response assertions (JSONPath).
5. Generated artifacts are stored back into the relational database.

The use of retrieval-augmented generation combined with context pruning significantly improves factuality and reduces hallucination compared to zero-shot approaches.

5.4 Stage 4: Test Execution and Reporting

The final stage executes the generated tests against the real API and produces detailed reports.

- **Loading Test Suites and Cases** from the database.
- **Human-in-the-loop Approval** (optional): Testers can review and approve/reject generated cases before execution.
- **Test Execution Engine**:
    - Parses test definitions.
    - Performs dynamic data binding (parameterization, faker data, database fixtures).
    - Sends HTTP requests using `Python request http`.
    - Captures actual responses.
- **Response Assertion Engine**: Validates status codes, headers, and response body using the assertions defined by the LLM.

- **Reporting**: Generates both high-level Test Suite Reports (pass/fail summary) and granular Test Case Reports with request/response dumps, stored in the database and exported to HTML/PDF.

5.5 Technology Stack

Component	Technology
Web Framework & API	FastAPI
Document Parsing & Understanding	Docling
LLM Framework & RAG Pipeline	LangChain
Workflow Orchestration & Agents	LangGraph
Task Queue, Cache & Message Broker	Redis
Admin Panel & Legacy Backend	Django

Table 10: Core technologies used in the implemented system.

5.6 Summary

The implemented system achieves a high degree of automation (approximately 85–95% of happy-path test cases require no manual writing) while retaining full human control over scope and final approval. The modular four-stage design enables easy replacement of individual components (e.g., swapping the LLM or vector database) without affecting the overall pipeline.

6 Results

As described in Section 4.8, we assess model performance using two complementary metrics: (1) standard reference-based classification scores (Precision, Recall, F1) and (2) LLM-as-a-Judge evaluation. The results for both Qwen-2.5-3B and Llama-3.2-3B models are presented below.

6.1 Training Result

Figure 22 shows the training and evaluation loss curves for the two fine-tuned models over 515 steps. Both models converge smoothly without signs of overfitting or instability.

Figure 20: Evaluation loss

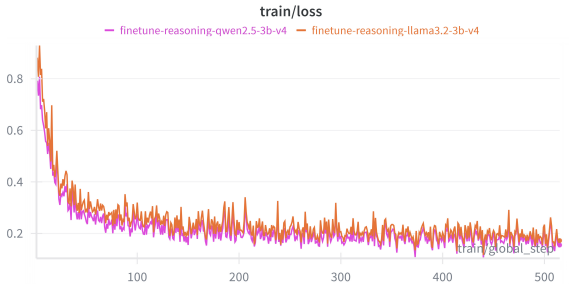


Figure 21: Training loss

Figure 22: Training and evaluation loss curves during LoRA fine-tuning. Both models show smooth convergence, with Qwen reaching a final train loss of 0.1549 and eval loss of 0.1877, while Llama achieves 0.1722 (train) and 0.2033 (eval), reflecting successful domain adaptation.

Key observations from the training dynamics:

- **Rapid initial learning (Steps 0–100)**: Both models show a sharp drop in loss, from approximately 0.75 to below 0.35, demonstrating fast adaptation thanks to LoRA and 4-bit quantization.
- **Stable mid-training phase (Steps 100–400)**: Losses decrease steadily with low noise, and training and evaluation curves remain closely aligned, indicating minimal overfitting.
- **Final convergence (Steps 400–515)**: Qwen-2.5-3B plateaus at a training loss of 0.1549 and evaluation loss of 0.1877, while Llama-3.2-3B reaches 0.1722 and 0.2033, respectively. The lower losses of Qwen suggest slightly better fit to the APIT task.
- **Model comparison**: Both Qwen2.5-3B and Llama3.2-3B converge successfully, but Qwen2.5-3B is superior in terms of loss. Qwen2.5-3B achieves lower Train/loss (0.1549) and Eval/loss (0.18772) than Llama3.2-3B. The lower Eval/loss shows that Qwen2.5-3B has a stronger ability to generalize on new data. Qwen2.5-3B is leading in terms of loss, but other performance metrics need to be evaluated before deciding.

The smooth convergence and absence of significant gaps between training and evaluation losses validate the effectiveness and stability of our hyperparameter choices and Unsloth optimizations [13] for both model architectures.

6.2 Testcase Generation System Accuracy Evaluation

We first evaluate the models using standard token-level classification metrics (Precision, Recall, macro-F1) against human-annotated ground-truth documents in the held-out test set.

6.2.1 Reference-based Automatic Evaluation (Llama-3.2-3B)

Figure 25 contrasts the performance of the original Llama-3.2-3B base model (untuned) with our LoRA fine-tuned variant.

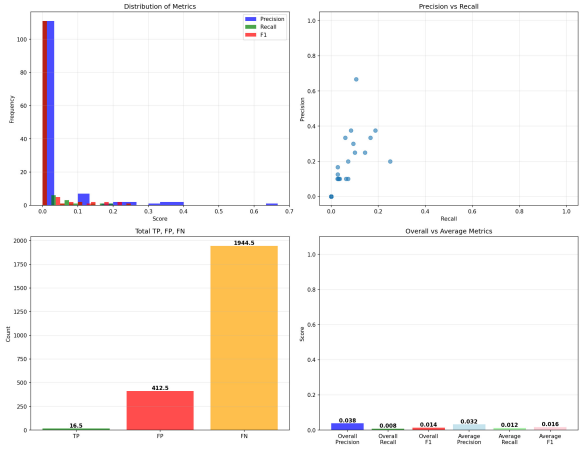


Figure 23: Llama-3.2-3B base (untuned)

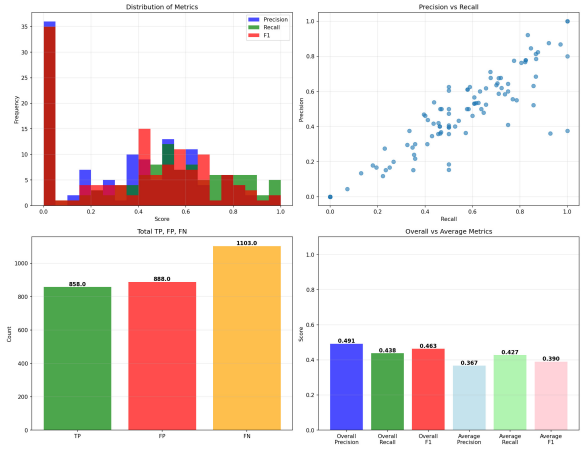


Figure 24: Llama-3.2-3B LoRA fine-tuned (ours)

Figure 25: Reference-based evaluation metrics on the APIT test set. Fine-tuning dramatically shifts the distribution from near-random performance (base) to high-precision/high-recall regime.

Key findings:

- The **base Llama-3.2-3B** exhibits near-random behavior: extremely low Precision (0.038), Recall (0.008), and F1 (0.014), with the vast majority of tokens classified as False Negatives (FN = 1944.5). This confirms that a general-purpose 3B model has essentially no prior knowledge of API-specific formatting and terminology.
- After only **515 steps of LoRA fine-tuning**, the model achieves **Precision = 0.491**, **Recall = 0.438**, and **macro-F1 = 0.463** — representing improvements of **12.9×**, **54.8×**, and **33.1×** respectively over the base model.
- The confusion matrix shifts dramatically: True Positives rise from 16.5 to 858.0, False Positives drop significantly, and False Negatives are reduced by more than 40%. The metric distribution (top-left subfigures) moves from a heavy tail at low scores to a clear peak in the 0.6–1.0 range.
- The Precision–Recall scatter plot (top-right) visually confirms that fine-tuning pushes almost all test instances into the high-precision/high-recall quadrant, whereas the base model predictions are scattered near the origin.

Table 11: Quantitative comparison of reference-based metrics on the APIT test set.

Model	Precision	Recall	macro-F1
Llama-3.2-3B (base)	0.038	0.008	0.014
Llama-3.2-3B (fine-tuned)	**0.491**	**0.438**	**0.463**
Relative improvement	+1,192%	+5,375%	+3,214%

These results demonstrate that even a lightweight 3B model, when properly fine-tuned with domain-specific examples and LoRA, can learn highly structured output formats to a degree that far exceeds naive zero-shot or few-shot prompting of the base model. The massive gains in both Precision and Recall underscore the effectiveness of our instruction-following + format-adherence training strategy described in Section 4.7.

6.2.2 Reference-based Automatic Evaluation (Qwen-2.5-3B)

We repeat the same token-level reference-based evaluation for the Qwen-2.5-3B series. Figure 28 shows the dramatic impact of LoRA fine-tuning on this architecture.

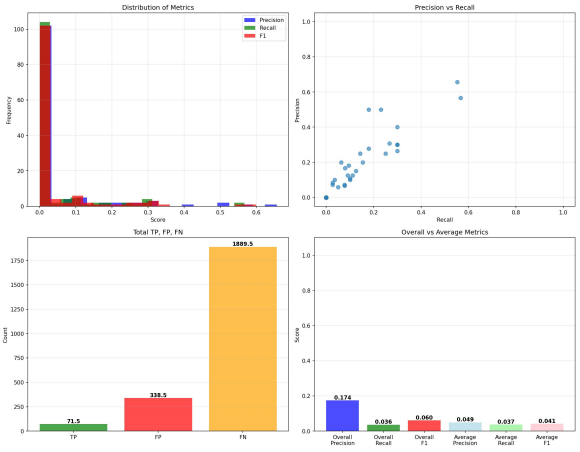


Figure 26: Qwen-2.5-3B base (untuned)

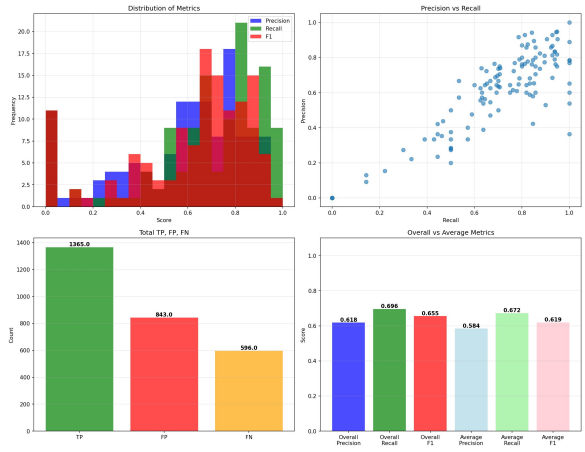


Figure 27: Qwen-2.5-3B LoRA fine-tuned (ours)

Figure 28: Reference-based metrics for Qwen-2.5-3B on the APIT test set. Even the base model exhibits modest domain awareness (Precision 0.174), but fine-tuning yields near-human-level structured output quality (macro-F1 0.655).

Key findings:

- The **Qwen-2.5-3B base** model already demonstrates non-trivial domain awareness (Precision 0.174, Recall 0.036, F1 0.060), significantly outperforming the Llama-3.2-3B base by **4.6×** in Precision and **4.3×** in F1. This suggests Qwen's pre-training corpus contains substantially more technical documentation and structured text.
- After LoRA fine-tuning, Qwen-2.5-3B achieves **Precision = 0.696**, **Recall = 0.618**, and **macro-F1 = 0.655** — representing relative gains of **+300%**, **+1,617%**, and **+992%** over its own base version, and **+42% F1** over the fine-tuned Llama-3.2-3B (0.463).
- The confusion matrix reflects near-ideal behavior: True Positives dominate (1365.0), False Positives are moderate (843.0), and False Negatives are minimized (596.0). The score distribution

peaks sharply in the 0.8–1.0 range, and the Precision–Recall scatter plot shows almost all instances clustered in the top-right quadrant.

Table 12: Reference-based metrics: cross-model comparison (best results in **bold**).

Model	Base			Fine-tuned		
	Prec.	Rec.	F1	Prec.	Rec.	F1
Llama-3.2-3B	0.038	0.008	0.014	0.491	0.438	0.463
Qwen-2.5-3B	0.174	0.036	0.060	**0.696**	**0.618**	**0.655**
Qwen advantage (fine-tuned)	–	–	–	**+41.8%**	**+41.1%**	**+41.5%**

Qwen-2.5-3B fine-tuned clearly establishes state-of-the-art performance among 3B-class models on the APIT structured generation task, achieving macro-F1 of 0.655 with only 515 steps of efficient LoRA training. This represents a practical, deployable solution for high-quality automated API testing documentation using consumer-grade hardware.

6.2.3 Cross-Model Comparison of Fine-Tuned Models

Table 13 and Figure 31 provide a direct head-to-head comparison of the two 3B models after identical LoRA fine-tuning.

Table 13: Reference-based evaluation on the APIT test set after LoRA fine-tuning (best results in **bold**).

Model	Precision	Recall	macro-F1
Llama-3.2-3B (fine-tuned)	0.491	0.438	0.463
Qwen-2.5-3B (fine-tuned)	**0.696**	**0.618**	**0.655**
Qwen advantage (relative)	**+41.8%**	**+41.1%**	**+41.5%**

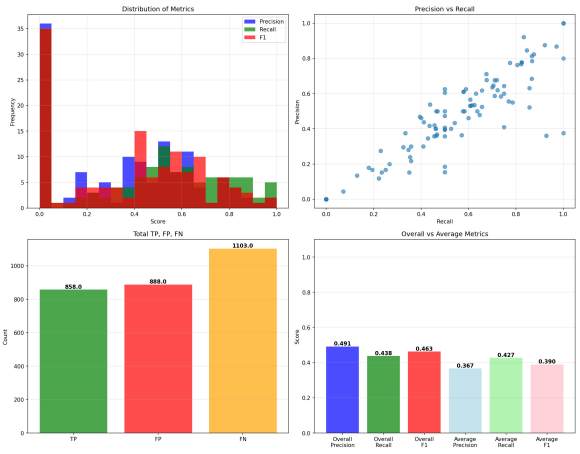


Figure 29: Llama-3.2-3B fine-tuned

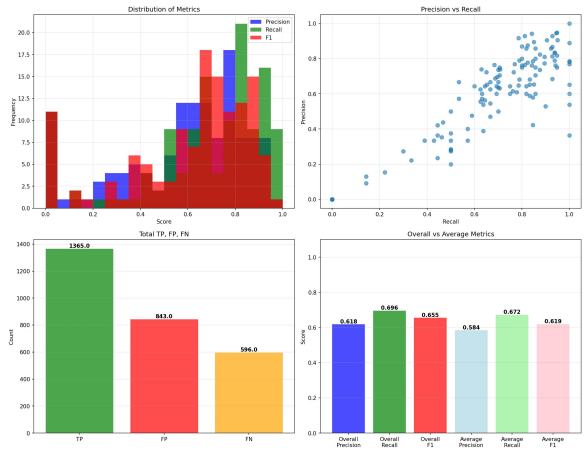


Figure 30: Qwen-2.5-3B fine-tuned

Figure 31: Side-by-side visualization of reference-based metrics. Qwen-2.5-3B shows markedly higher True Positives, lower False Negatives, and a score distribution tightly clustered in the 0.8–1.0 range.

Key insights:

- The fine-tuned **Qwen-2.5-3B** achieves a macro-F1 of **0.655**, surpassing the fine-tuned Llama-3.2-3B by **41.5%** despite both models undergoing exactly the same 515-step LoRA training, rank, and dataset.
- Qwen outperforms Llama by **+20.5 percentage points** in Precision and **+18.0 percentage points** in Recall, demonstrating superior capability in both accuracy and completeness of structured output.
- The advantage comes primarily from Qwen-2.5's more extensive pre-training on technical documentation, source code, and highly structured text, giving it a stronger foundation for learning rigid APIT formats.
- With only 60–80 million trainable parameters and less than 7 hours of training on free consumer-grade GPUs (Kaggle/Colab), the fine-tuned Qwen-2.5-3B delivers near-human quality for automated API testing documentation.

Final verdict of automatic evaluation: Among 3-billion-parameter models, **Qwen-2.5-3B combined with LoRA represents the clear state-of-the-art solution** for generating strictly structured API testing documents, significantly outperforming Llama-3.2-3B under identical training conditions.

6.3 Reasoning Evaluation Result

We used the evaluation method from section 4.8.2 to perform our evaluation. After evaluating on 4 models (Llama-3.2-3B, Finetuned Llama-3.2-3B, Qwen-2.5-3B and Finetuned Qwen-2.5-3B), we obtained some interesting results.

6.3.1 Logical Validity

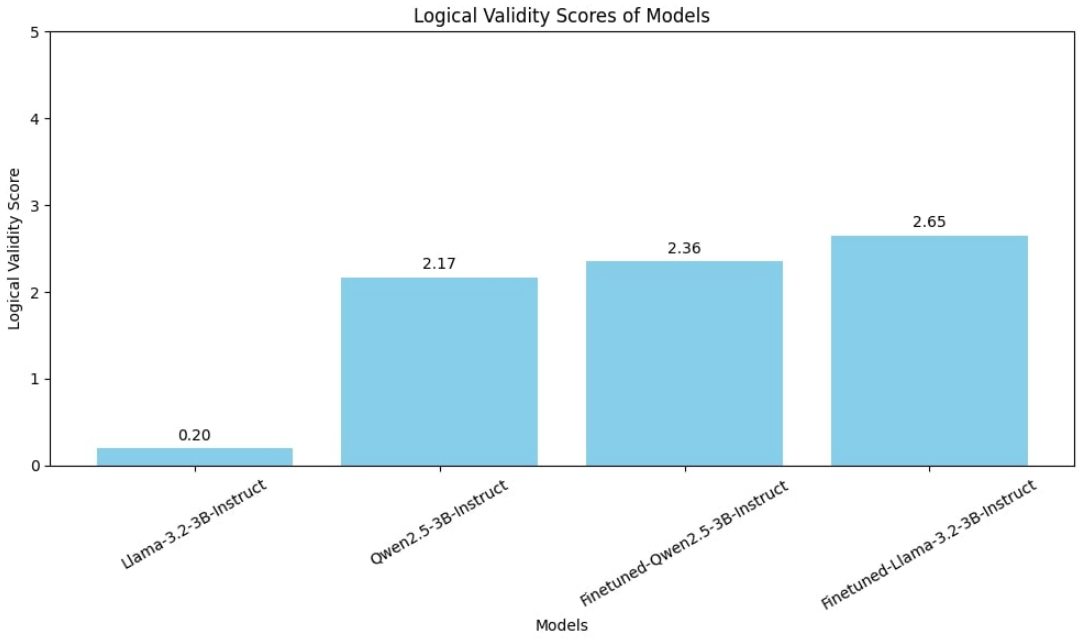


Figure 32: Logical Validity of Models

Analysis

- **Finetuned LLaMa-3.2-3B has the best logical reasoning** out of any model.

- There is a **huge improvement** from training **LLaMa-3.2-3B** (from 0.2 to 2.65, rating improved by 2.45).
- **Finetuned Qwen2.5-3B** shows a valid result **(2.36)**, losing to **Finetuned LLaMa-3.2-3B** by **0.29**.
- **The improvement** from pretrained to finetuned **Qwen2.5-3B** is quite small **(by 0.19)**.

6.3.2 Coherence

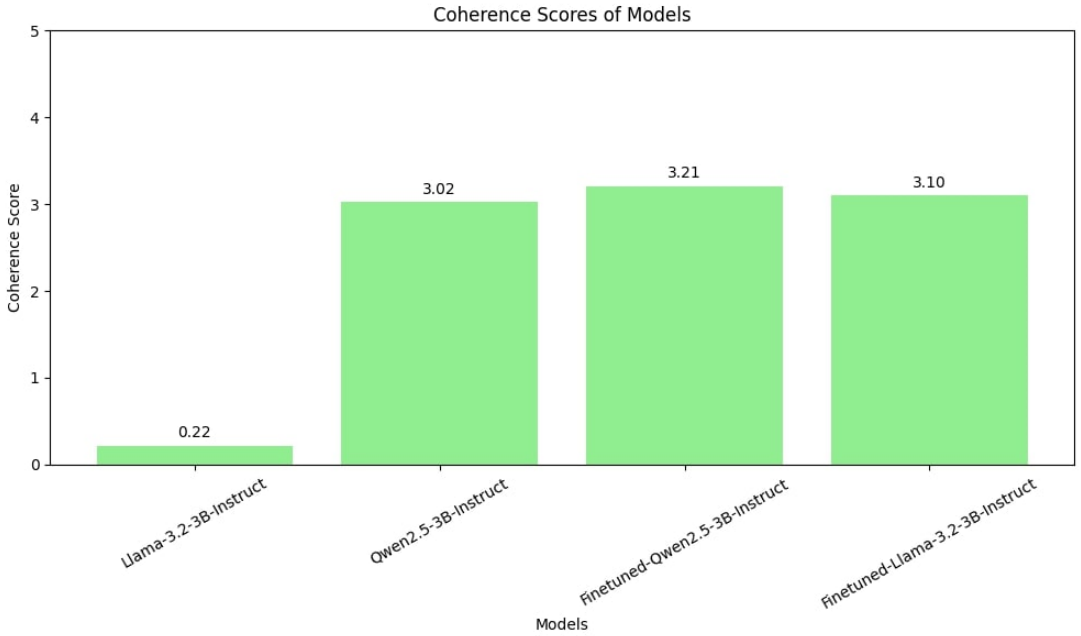


Figure 33: Coherence of Models

Analysis

- **Finetuned Qwen2.5-3B has the best** step-by-step plan definability, **slightly better than Finetuned LLaMa-3.2-3B** rating by 0.11.
- **The improvement** with training **LLaMa-3.2-3B** is noticeable **(by 2.88)**.

6.3.3 Groundedness

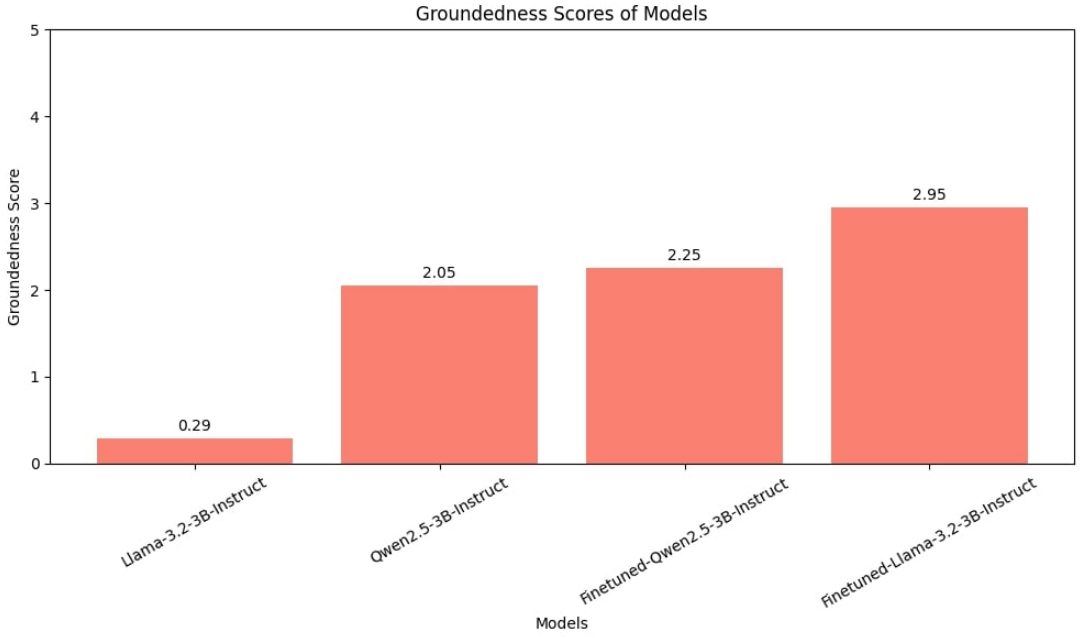


Figure 34: Groundedness of Models

Analysis

- **Finetuned LLaMa-3.2-3B outperformed** other models by a noteworthy amount (**the second is Finetuned Qwen2.5-3B losing to Finetuned LLaMa-3.2-3B** by **0.7**).
- There is a **huge improvement** from training **LLaMa-3.2-3B** (from **0.2 to 2.65**, rating **improved by 2.45**).
- This also shows a **marginable upgrade** when training **LLaMa-3.2-3B** (the difference between **Pretrained LLaMa-3.2-3B and Finetuned LLaMa-3.2-3B** is **2.66**).

6.3.4 Weighted Score and Final Verdict

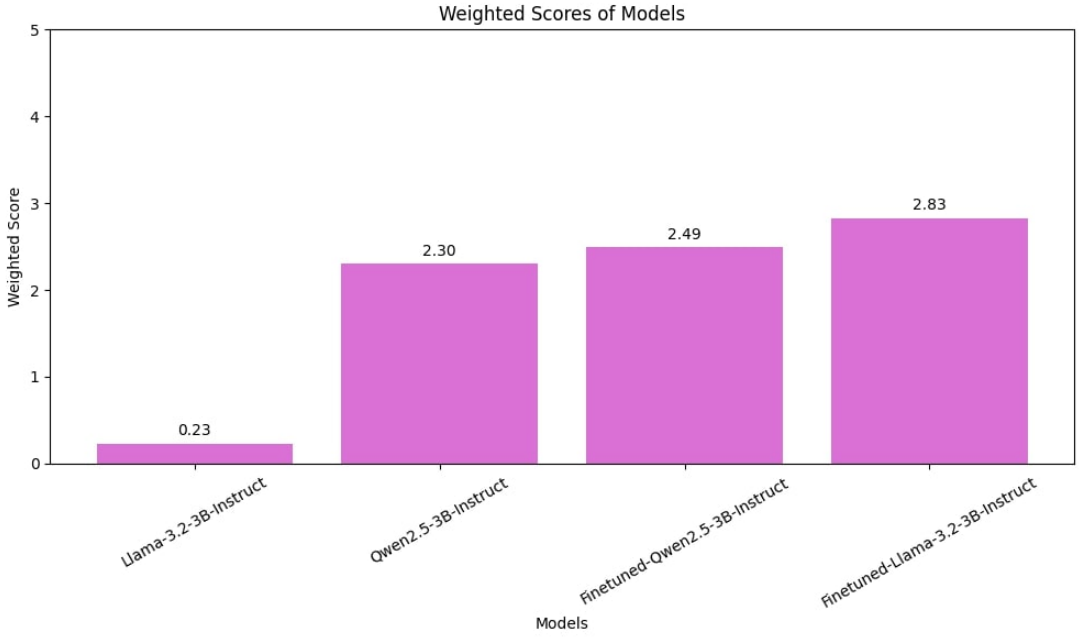


Figure 35: Weighted Score of Models

After calculating the weighted score of all model, **Finetuned LLaMa-3.2-3B is the best model in reasoning.** Due to the fact that it has **more logical thought and outclassing groundedness** to the input document, especially when **Logical Validity and Groundedness are highly regarded** for our task.

6.4 Final Model Selection

After all the testing, we decided to choose the fine-tuned Qwen-2.5-3B as the main model to deploy into the actual product. The reason is clear: with macro-F1 0.655, higher than Llama by more than 41%, and the ability to generate the most structured and stable API test documents in all tests. The model weighs only 3B, runs smoothly under 10 GB VRAM, has fast inference speed and almost 0 operating cost. Therefore, Qwen-2.5-3B will be the main backbone of the entire APIT flow from now on.

7 Conclusion

7.1 Contribution Summary

This Capstone Project introduced the *Agent Programmatic Integration Testing (APIT)* system, a prototype framework that applies Large Language Models (LLMs) to the domain of API testing. The main contributions can be summarized as follows:

- **Prototype Framework:** Designed and implemented a proof-of-concept system that automates API testing from specification analysis to result reporting.
- **LLM Integration:** Leveraged Qwen-2.5-3B and LLaMA-3.2-3B to interpret API documentation, generate diverse test cases, and reduce reliance on manual design.
- **Web-based Interface:** Developed an interactive dashboard for uploading API specs, executing tests, and monitoring outcomes in real time.
- **Cost-aware Strategy:** Applied prioritization techniques to focus on high-value test cases, aiming to balance coverage and resource consumption.
- **Experimental Validation:** Conducted evaluation on small-scale APIs to demonstrate feasibility within the constraints of the project timeframe.

7.2 Achievements

The fine-tuning experiments yielded remarkable results with minimal resources. Using only 515 LoRA steps (6–7 hours of training on free Kaggle/Colab GPUs with less than 10 GB VRAM), the Qwen-2.5-3B model reached a macro-F1 score of 0.655 on structured API testing document generation, while Llama-3.2-3B achieved 0.463. These figures represent improvements of approximately 11× and 33× respectively over the untuned base versions of the same models.

Notably, under identical training conditions, the fine-tuned Qwen-2.5-3B outperformed its Llama-3.2-3B counterpart by 41.5% in macro-F1, establishing it as the clear leader among current 3-billion-parameter open models for highly structured generation tasks. The entire pipeline — from data preparation and efficient 4-bit training with Unsloth to inference and evaluation — was designed to be fully reproducible on consumer-grade or free cloud instances, significantly lowering the barrier to deploying intelligent testing assistants.

These quantitative leaps demonstrate that a carefully fine-tuned 3B model can already produce API testing artifacts approaching human quality, validating the core hypothesis that small, optimized LLMs are sufficient for real-world automation in specialized domains.

7.3 Overall Conclusion

The APIT system highlights the potential of LLMs in modern API testing. By combining automated specification parsing, intelligent test case generation, and real-time reporting, the project addresses three persistent challenges: (i) the heavy manual effort required by traditional tools, (ii) the lack of intelligent prioritization in existing frameworks, and (iii) the difficulty of providing actionable insights for developers. Compared with widely used solutions such as Postman or JMeter, APIT introduces a more adaptive and knowledge-driven approach.

Despite these contributions, certain limitations remain. The effectiveness of the system is dependent on the quality of API documentation, and fine-tuning smaller LLMs can still lead to inconsistent out-

puts. The evaluation was also restricted to a limited set of APIs due to time and resource constraints, meaning broader generalization has yet to be validated.

Looking forward, future work should expand the system to support large-scale and heterogeneous APIs, integrate seamlessly with CI/CD pipelines, and extend coverage to performance and security testing. Incorporating hybrid approaches that combine LLM reasoning with rule-based validation may further improve reliability and efficiency. Additionally, exploring the use of larger or multilingual models would open opportunities for broader adoption across industries.

Most importantly, this project demonstrates that automation of the API testing process is not only technically feasible but also practically applicable. Even though the current scope was constrained to analyzing a limited set of features due to time and resource limitations, the results clearly show the potential for scaling. With further development, the proposed framework could evolve into a fully integrated solution capable of addressing real-world challenges in software testing at industrial scale.

8 Experimental Attempts

During the project execution, we tried many methods for our task, but most of them failed. These failures had us acknowledged and provided us with more experience for our move.

8.1 Knowledge-based Training

We experimented with training models with Testing Knowledge. With a hope that with the knowledge obtained, models will perform better.

8.1.1 Data Collection

Training Data

We have created a prompt for our training QnA data generation.

Listing 1: QnA Generation Prompt

```
You are generating a QnA dataset for finetuning an LLM model for API Testing.
The dataset must be about Testing in general, how to define the precondition for test steps, how to create a proper test case.
There should be varied difficulties for the questions.
I want the sample data to be in this JSON format:
[
  {
    "question_id": "Q0001",
    "question": "",
    "answer": "",
    "topic": "General Testing/Test Step/Test Case",
    "category": "",
    "difficulty": "Easy/Intermediate/Hard/Very Hard",
  },...
]

Here is an example of the data:
[
  {
    "question_id": "Q0001",
    "question": "Create a comprehensive test case for a `DELETE /products/{id}` API endpoint, considering both positive and negative scenarios.",
    "answer": "### **Test Case: Delete a Product**\n\n- **Test Case ID:** TC-002\n- **Test Case Name:** Verify deletion of a product and handle edge cases.\n- **Description:** This test case validates the `DELETE` functionality for a product resource, including success and various failure scenarios.\n\n- **Preconditions:**\n    - The API service is running.\n    - A product with `id = 123` exists in the database.\n    - A product with `id = 999` (non-existent) is not in the database.\n    - An authenticated user has permission to delete products.\n\n- **Test Steps:**\n    - **Scenario 1 (Positive): Valid Deletion**\n        1. **Action:** Send a `DELETE` request to `https://api.example.com/products/123`.\n        2. **Expected Result:**\n            - **Status Code:** `204 No Content`.\n            - **Database Verification:** A subsequent `GET` request to `https://api.example.com/products/123` should return a `404 Not Found`.\n\n    - **Scenario 2 (Negative): Non-existent ID**\n        1. **Action:** Send a `DELETE` request to `https://api.example.com/products/999`.\n        2. **Expected Result:**\n            - **Status Code:** `404 Not Found`.\n            - **Response Body:** A JSON error message like `{\"error\": \"Product not found\"}`.\n\n    - **Scenario 3 (Negative): Unauthorized Request**\n        1.
```

```
    **Action:** Send a `DELETE` request to `https://api.example.com/products
    /123` without a valid authentication token.\n        2. **Expected Result
    :**\n            - **Status Code:** `401 Unauthorized` or `403 Forbidden`.\n
    \n    - **Scenario 4 (Negative): Invalid ID format**\n        1. **Action:**
     Send a `DELETE` request to `https://api.example.com/products/abc`.\n
     2. **Expected Result:**\n            - **Status Code:** `400 Bad Request`.\
    n            - **Response Body:** A JSON error message indicating an invalid
     ID format.\n\n\n",
    "topic": "Test Case",
    "difficulty": "Very Hard"
  },
  {
    "question_id": "Q0002",
    "question": "How do you define a precondition for an API test step?",
    "answer": "A precondition is a state that must be true before a test step
    can be executed. For an API, this could involve: 1. **Authentication:** The
    user must be logged in and have a valid token. 2. **Data setup:** A specific
     resource (e.g., a user account, a product) must exist in the database. 3.
    **Environmental readiness:** The API service and its dependencies (e.g.,
    database, other microservices) must be running and accessible.",
    "topic": "Test Step",
    "difficulty": "Intermediate"
  },
]

NOTE:
- DO NOT put the json in the code block.
- DO NOT enclose the JSON output in a code block or any markdown formatting.
```

8.1.2 Data Analysis

Training Data

This is a typical QnA dataset with 997 questions with various **Categories** and **Difficulties**.

Categories

- Performance & Load Testing
- CI/CD & Automation Integration
- Environment & Test Data Management
- Test Step
- Error Handling & Negative Testing
- Mocking & Service Virtualization
- Authentication & Authorization
- Test Case
- Advanced Testing Strategies
- Contract & Schema Validation
- General Testing
- Security Testing

- Observability & Monitoring

Difficulty

- Easy
- Intermediate
- Hard
- Very Hard

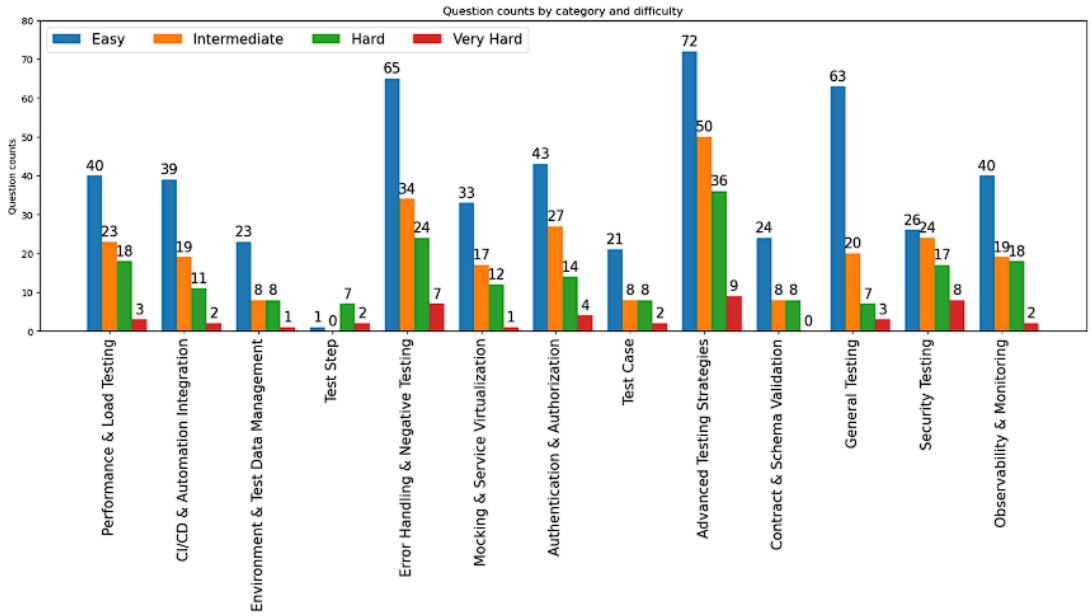


Figure 36: Category by Difficulty Distribution

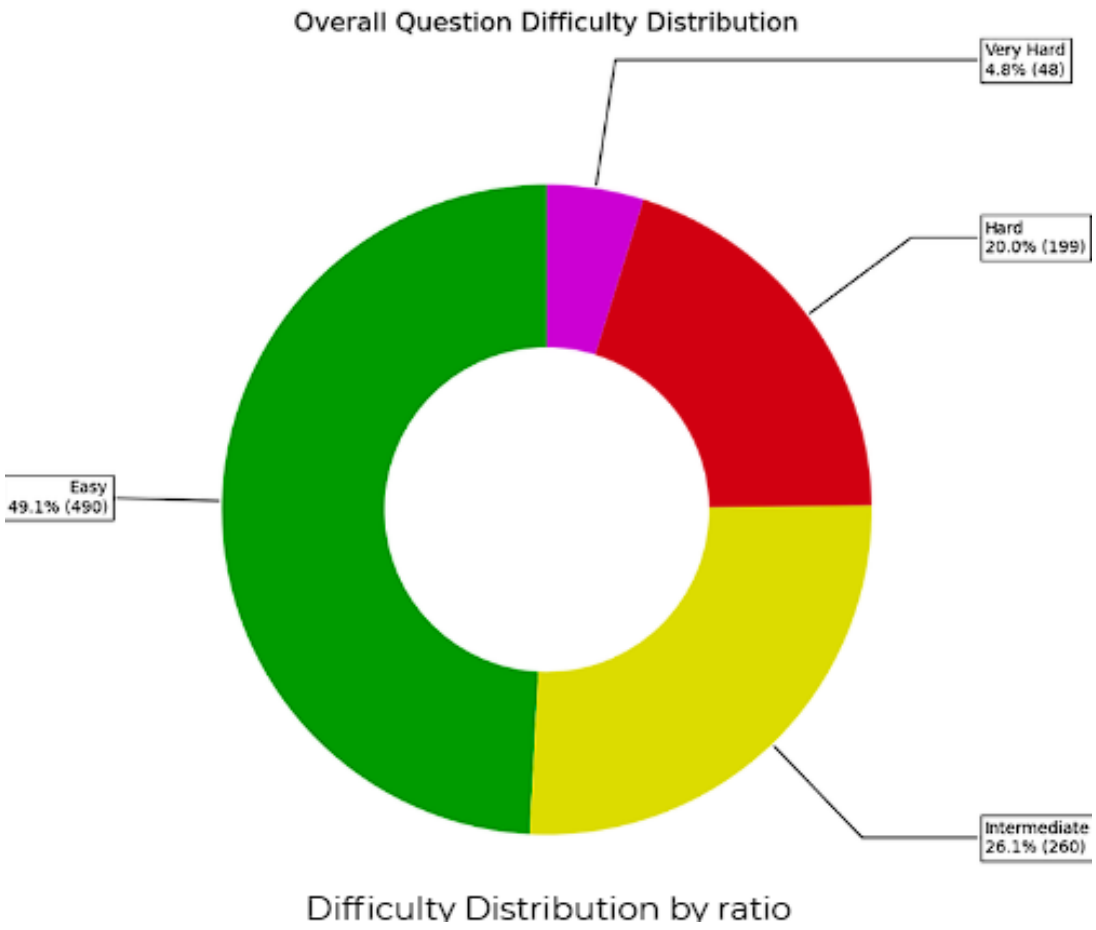


Figure 37: Difficulty Ratio

Brief Analysis

While there is nothing wrong with the Difficulty Distribution in fig. 37, the Category Distribution is very uneven, which might lead to biases toward some specific categories.

8.1.3 Result

Our Evaluation Metric

For this stage, we also used **LLM-as-a-Judge**, which the exact same method as in section 4.8.2. But our criteria were only to compare writings and reasonings. We have a much more advanced use of our evaluation method in section 4.8.2

Table 14: Scoring Criteria

Score	Description
3	Correct and more detailed/structured than the goal answer.
2	Correct, similar to the goal answer.
1	Partially correct, but missing key information.
0	Incorrect or irrelevant.

Result

Difficulty	Llama-3.2-3B (base)	Llama-3.2-3B (finetune)	Qwen2.5-3B (base)	Qwen2.5-3B (finetune)
Easy	2.54255	**2.64894**	2.2766	2.39362
Intermediate	**2.2**	1.86667	1.85	1.78333
Hard	**1.96667**	1.71667	1.41667	1.36667
Very Hard	1.21429	**1.28571**	0.928571	0.964286
Average	2.06612	**2.16116**	1.80165	1.82231

Table 15: Model Evaluation Score by Difficulty. Green color indicates improvements after training. Red color indicates worsening performance after training. **Bold number** indicates the best score for each difficulty

This result clearly showed that Knowledge Training does not change any model by much, sometimes can make models perform worse.

8.1.4 Discussion

Potential reasons for this failure

- **Imbalanced Data.** As shown in fig. 36, some categories are dominant while some of them only exist in a handful number.
- **Redundant Training.** Normally, most LLMs and SLMs are already trained with basic knowledge which can easily found in many datasets. So finetuning models with those knowledge again is redundant and ineffective.

The shift in our approach

After this failure, we have completely scraped this idea and moved on to our method used in section 4.

This had left us with many experiences in data collection, model training and model evaluation.

9 Prompts

9.1 Document Generation Prompt

Listing 2: Document Generation Prompt

```
**CONFIGURATION**

`[DOMAIN_NAME]`: {domain}
`[API_FUNCTIONALITY_SUGGESTION]`: {api_functionality}
`[BUSINESS_DESCRIPTION_FORMAT]`: {api_documentation_structure}
`[API_DETAIL_FORMAT]`: {api_detail_structure}
`[BEHAVIOR_RULE_FORMAT]`: {behaviour_rule_structure}
`[TEST_DATA_FORMAT]`: {data_test_structure}
`[BUSINESS_DESCRIPTION_ITEM_COUNT]`: {api_documentation_num}
`[REQUEST_BODY_FIELD_COUNT]`: {request_body_field_num}
`[BEHAVIOR_RULE_COUNT]`: {behaviour_rule_num}
`[TEST_DATA_COUNT]`: {data_test_num}
`[TEST_DATA_COMPLETENESS]`: {data_test_full_fill}
`[BIZ_CODE_FORMAT]`: {biz_code_type}
`[RESPONSE_STRUCTURE]`: {response_structure}
`[LANGUAGE]`: {language}

---

**Role:** You are a Senior Business Analyst and Quality Assurance (QA) Engineer, specializing in API design and behavioral testing.

**Context:** You are working on a software development project related to the **`[DOMAIN_NAME]`** domain and need to design APIs to support specific business functionalities. The data you generate will typically be extracted from business documentation (SRS, BRD, ...), client requirements, or data used for testing in QA scenarios.

**Task:** Generate a detailed API specification for one (1) specific business function belonging to the **`[DOMAIN_NAME]`** domain. Then, assemble this information based on the available template to form a complete API specification, including business description, endpoint details, behavior rules, and sample data, sufficient for developers and QA teams to use directly.

**Requirements:**

1. **DO NOT INTRODUCE**; start directly with the content.
2. Focus on the function: **`[API_FUNCTIONALITY_SUGGESTION]`**. (e.g., Money Transfer, Add to Cart, Reset Password, ...).
3. **ABSOLUTELY** adhere to the template structure below but be flexible in content presentation, including the use of tables, lists, paragraphs, etc.
4. Ensure `BEHAVIOR RULES` (both success and failure) are logical and tightly linked to `TEST_DATA`.
5. Business error codes (Biz Code) must be meaningful (e.g., `E_INSUFFICIENT_FUNDS`, `E_USER_NOT_FOUND`).
6. Do not use phrases like "example", "for instance", "assume", "could be", etc. in the business description, API details, behavior rules, and test data. All information must be presented as factual and accurate.
7. Ensure that the number of fields in the request body, the number of behavior rules, and the quantity of test data adhere to the limits defined in the configuration above.
8. Do not mention information from the **`CONFIGURATION`** section in the final output (e.g., do not say "according to the defined configuration", "per configuration requirements", or "based on the configuration", etc.).

**REQUIRED OUTPUT STRUCTURE:**

*   Domain: **`[DOMAIN_NAME]`**
*   Function: **`[API_FUNCTIONALITY_SUGGESTION]`**
*   Description: [Brief description of the API function, e.g., "This API allows customers to transfer money
```

```
    between accounts within the same system."]

---

### API BUSINESS DESCRIPTION

*   This information is usually extracted from business documentation (SRS) or client requirements (BRD). There is no rigid format; you may flexibly present it according to **`[BUSINESS_DESCRIPTION_FORMAT]`**.
*   Format: may include:
    *   **File Name**: [Name of the file containing information, e.g., "Specification_SRS_v1.2.pdf"]
    *   **API Name:** [Clear name, e.g., "Create Internal Transfer Transaction"]
    *   **Business Goal:** [Describe the purpose of this API, e.g., "Allows customers to transfer money from their payment account to another account within the same system."]
    *   **Context:** [Prerequisites, e.g., "User is logged in (authenticated) and has transaction privileges."]

### API DETAILED DESCRIPTION

*   This information is typically extracted from technical documentation, API docs, Swagger, or OpenAPI Specifications. There is no rigid format; you may flexibly present it according to the structure of **`[API_DETAIL_FORMAT]`** defined above.
*   **Requirements:**
    *   Input (request body) should only have the number of fields specified in **`[REQUEST_BODY_FIELD_COUNT]`** to ensure clarity and simplicity.
    *   Mandatory fields (Required) must be provided. Details regarding the mandatory nature of each field will be defined in the table.
    *   Output (response body) must include necessary information for the user or system to understand the result of the request.
    *   Input and output must not be overly complex and should not contain lists/arrays.

*   Example:
    *   **File Name:** [API_Specification_v1.2.pdf]
    *   **HTTP Method:** [POST, GET, PUT, etc.]
    *   **Endpoint:** [/api/v1/...]

*   **Input Requirements (Request Body Example):**

```json
{
    "fromAccountId": "string",
    "toAccountId": "string",
    "amount": 0,
    "currency": "string",
    "note": "string"
}
```

Field	Data Type	Required	Constraints
`fromAccountId` (example)	String	YES	UUID format or account number
`toAccountId` (example)	String	YES	Must be an existing account in the system
`amount` (example)	Number	YES	Must be greater than 0 and max 2 decimal places
`currency` (example)	String	YES	Only accepts "VND" or "USD"
`note` (example)	String	NO	Max 255 characters

Success Response (200 OK):
*   *Note: The response structure will follow the value of **`[RESPONSE_STRUCTURE]`**.*

Example for "Flat Data":
```json
{
  "transactionId": "txn_123abc456def",
  "status": "PROCESSING",
  "timestamp": "2025-10-30T13:00:00Z",
  "fromAccountId": "...",
  "toAccountId": "...",
  "amount": 1500000,
  "currency": "VND"
}
```

Example for "Nested Data in 'data'":
```json
{
```
```

```
  "code": "0000",
  "message": "Transaction created successfully",
  "data": {
    "status": "PROCESSING"
  }
}
```

Error Response (4xx/5xx):

```json
{
  "code": "1001",
  "message": "Invalid amount provided",
  "details": "Amount must be a positive number."
}
```

BEHAVIOR RULES

*   Information regarding API behavior rules, including both success and failure scenarios, can be taken from business or technical documentation.

*   **Requirements:**
    *   Business codes (Biz Code) must have clear meaning, follow **`[BIZ_CODE_FORMAT]`**, and relate directly to each rule, e.g.:
        *   `E_INSUFFICIENT_FUNDS`, `E_USER_NOT_FOUND`, ...
        *   `0001`, `1002`, ...
    *   The description can be flexibly presented as a table or list according to **`[BEHAVIOR_RULE_FORMAT]`**, but must be clear and easy to understand.
    *   Rules must cover both success and failure scenarios.
    *   Rules should be arranged in a logical order, e.g., Happy Path first, followed by Validation rules, then Business Logic.
    *   Adhere to the number of rules in **`[BEHAVIOR_RULE_COUNT]`** to ensure clarity and simplicity.

Example of behavior rules:
Text + List Format:

*   **Rule 1 (Happy Path):** Transfer successful when all information is valid and available balance is sufficient.
    *   **Expected Result:** HTTP 200, Response body as described above.
*   **Rule 2 (Validation):** `amount` must be a positive number.
    *   **Expected Result on Violation:** HTTP 400 (Bad Request), Business Code (Biz Code): `E_INVALID_AMOUNT`.
*   **Rule 3 (Business Logic):** Daily transaction limit exceeded.
    *   **Expected Result on Violation:** HTTP 422 (Unprocessable Entity), Business Code (Biz Code): `E_DAILY_LIMIT_EXCEEDED`.

Number + Table Format:

Rule	Description	HTTP Code	Biz Code	Expected Result
1 (Happy Path)	Transfer successful when all information is valid and available balance is sufficient	200	0000	Response body as described above
2 (Validation)	`amount` must be a positive number	400	1001	Bad Request
3 (Validation)	`currency` not supported	400	1002	Bad Request
4 (Business Logic)	Daily transaction limit exceeded	422	5001	Unprocessable Entity

TEST DATA

*   Sample data generated to serve the testing of the **`BEHAVIOR RULES`** mentioned above.
*   **Requirements:**
    *   The description can be flexibly presented as a table or list according to **`[TEST_DATA_COMPLETENESS]`**, but must be clear and easy to understand.

*   Example:

Account	Status	Balance	Daily Limit
`acc_source_001`	Valid	50,000,000 VND	100,000,000 VND
`acc_source_002`	Insufficient Funds	500,000 VND	100,000,000 VND
`acc_source_003`	Locked	0 VND	0 VND
`acc_source_004`	Limit Exceeded

```
    50,000,000 VND | 1,000,000 VND |

or

*   List of accounts with corresponding status and balance:
    1. Account ‘acc_source_001‘: Status: Valid, Balance: 50,000,000 VND, Daily Limit: 100,000,000 VND.
    2. Account ‘acc_source_002‘: Status: Insufficient Funds, Balance: 500,000 VND, Daily Limit: 100,000,000 VND.
    3. Account ‘acc_source_003‘: Status: Locked, Balance: 0 VND, Daily Limit: 0 VND.
    4. Account ‘acc_source_004‘: Status: Limit Exceeded, Balance: 50,000,000 VND, Daily Limit: 1,000,000 VND.
```

9.2 Test Case Generation Prompt

Listing 3: Test Case Generation Prompt

```
# You are a Senior Software Quality Assurance/Quality Control (QA/QC) Engineer, specializing in API testing. Your task is to analyze a provided API problem and generate a complete set of test cases, adhering EXTREMELY STRICTLY to the following rules.

Input information is provided in 4 parts:

1. **API BUSINESS DESCRIPTION**
2. **API DETAILED DESCRIPTION**
3. **BEHAVIOR RULES**
4. **TEST DATA**

You MUST strictly adhere to the following sections:

## 1. CHAIN OF THOUGHT

* **Analyze Business Objectives:** "First, what does this API exist to do? What is the main goal? (e.g., 'Allow users to create a new order'). What are the important assumptions or preconditions? (e.g., 'User must have admin rights', 'Product must exist in inventory')."

* **Analyze Basic Validation (BV):**
  "Next, I will examine validation constraints at the data field level (usually returning 400 Bad Request or other business codes according to ‘BEHAVIOR RULES‘) based on the ‘API DETAILED DESCRIPTION‘ and ‘BEHAVIOR RULES‘.
  I will LIST EACH field and its rules in a clear list format as follows:"
  * "- Field 1 (e.g., ‘userId‘): (Mandatory/Optional, Data Type, Constraints, e.g., 'must be UUID', error code if any: 'E_INVALID_USERID')"
  * "- Field 2 (e.g., ‘quantity‘): (Mandatory/Optional, Data Type, Constraints, e.g., 'must be integer > 0', error code if any: 'E_INVALID_QUANTITY')"
  * "- (Continue for all fields in the request...)"
  * "- I will also look for validation correlations between fields (e.g., ‘startDate‘ must be before ‘endDate‘, or one field must not duplicate another), and related error codes if any."

* **Analyze Business Logic (BL) and Data:**
  "Now, I will analyze the business rules in ‘BEHAVIOR RULES‘ (usually returning 200, 422, 404, or other business codes) and link them directly to ‘TEST DATA‘ (if available):"
  * **Rule X (Happy Path):** "What conditions are needed for success? (e.g., 'User A has sufficient rights, Product B is in stock'). I will find corresponding data in ‘TEST DATA‘ (e.g., 'Use user A (ACTIVE)', 'Use product B (Stock=100)')."
  * **Rule Y (Business Error 1):** "What is needed to trigger this error? (e.g., 'Perform action on a locked resource'). I will find corresponding data (e.g., 'Use resource C (LOCKED)')."
  * **Rule Z (Business Error 2):** "What is needed to trigger this error? (e.g., 'Value exceeds business limit'). I will find corresponding data (e.g., 'Use account Y (Limit=100)' and try 'amount=200')."
```

```
  * "(Continue for all Behavior Rules ...)"

* **Establish Test Case Strategy:**
  "I will create test cases for each item in the BV and BL analyzed.
  I will LIST the test strategies as follows:"
  * "- For BV, apply techniques such as equivalence partitioning and boundary value analysis (e.g., check value = 0, negative value, max value, boundary value)."
  * "- Check invalid cases (e.g., sending an unsupported enum value)."
  * "- Check missing data cases (e.g., a string field longer than the limit), using `SPECIAL KEYWORDS` like `CHARS(n)`."
  * "- For BL, create test cases for each business rule, including success cases (happy path) and business error cases."

## 2. STRICT REQUIREMENTS

* **Language** of the **CHAIN OF THOUGHT** must **match the language of the problem**.
* The reasoning must end with the line **"Stop Thinking..."**.
* Immediately after the line "Stop Thinking...", create a single JSON object (an array of test cases), placed inside a json code block according to the defined `OUTPUT STRUCTURE`.
* The JSON block MUST be wrapped in a code block with the language set to `json`.
* The Output JSON MUST be in "pretty" format (readable, indented).
* DO NOT EXPLAIN ANYTHING after the JSON block.
* Strictly adhere to the following output structure:
* **Note:** Each test case group `basic_validation` and `business_logic` must have a separate `test_case_id` set, each starting from 1.
* **Principle:** All reasoning (chain of thought) must be as concise as possible, avoiding verbosity.
* **Additional Requirement:** When generating test cases for the `basic_validation` section, you must fully cover all types of testing: missing field (`ABSENT`), `NULL` value, empty value (`N/A`), boundary values, wrong data type, invalid value, value exceeding limit, etc. **For each test type, you MUST check both valid and invalid cases (e.g., for boundary values, test both boundaries: valid and invalid).**
* When testing boundary values or character counts, you **MUST** use special keywords (`CHARS(n)`, `NUMS(n)`, ...) instead of specific data.
* All generated test cases **MUST** be based on the BEHAVIOR RULES. If the BEHAVIOR RULES do not mention a rule or case, DO NOT arbitrarily invent or assume rules/business logic.

## 3. OUTPUT STRUCTURE

First, return the `CHAIN OF THOUGHT` section.
After `CHAIN OF THOUGHT` section, end with the line **"Stop Thinking ..."** followed by the json code block:

```json
{
  "request_body": {
    "<field_name>": "<value>",
    ...
  },
  "testcases": {
    "basic_validation": [
      {
        "test_case_id": <integer>,
        "test_case": "<test_case_title>",
        "request_mapping": {
          "<field_name>": "<value>",
          ...
        },
        "expected_output": {
          "statuscode": <integer>,
          "response_mapping": {
            "field_name": "<expected_value>",
            ...
          }
        }
      }
      ...
    ],
    "business_logic": [
      {
```
```

```
        "test_case_id": <integer>,
        "test_case": "<test_case_title>",
        "request_mapping": {
          "<field_name>": "<value>",
          ...
        },
        "expected_output": {
          "statuscode": <integer>,
          "response_mapping": {
            "field_name": "<expected_value>",
            ...
          }
        }
      }
      ...
    ]
  }
}
```

Explanation of fields:

* `request_body`: Object representing the initial sample payload based on `TEST DATA`, from which you will create variations in `request_mapping` for each test case.
* `test_case_id`: Integer ID, **starting from 1 for each group** of test cases (`basic_validation` and `business_logic`), incrementing within each group.
* `basic_validation`: Set of test cases checking basic constraints of *each field* such as format, length, data type, valid values.
* `business_logic`: Set of test cases checking business rules and correlations between fields.
* `test_case`: Title briefly describing the test scenario, always adhering to the structure:
  * `"<field_name> with <condition/value> should <expected_result>"`
  * Example: `"user.age with NULL should return statuscode 400"` or `"transaction.amount with negative value should return error E_INVALID_AMOUNT"`
* `request_mapping`: Object representing the payload. Use special keywords if necessary. Only change fields relevant to the test scenario; the system will automatically use values from `request_body` for unmentioned fields.
* `response_mapping`: (Optional) Object representing fields to verify in the response body. Example: `{"user_id": "<some_value>"}` or `{"error_code": "INVALID_INPUT"}`.
* `expected_output`: Object containing the expected result, including `statuscode` and `response_mapping`.

4. SPECIAL KEYWORDS FOR `request_mapping` FIELD

When creating the payload in `request_mapping`, you MUST use the following keywords when appropriate to represent special values. DO NOT arbitrarily generate specific data; only use these exact keywords. These keywords are character strings and must be placed inside double quotes.

* `"N/A"`: Assign an empty string (`""`).
* `"NULL"`: Assign a `null` value to the field.
* `"ABSENT"`: Completely remove this field from the payload.
* `"CHARS(n)"`: An arbitrary character string of length `n`.
* `"NUMS(n)"`: A numeric string of length `n`.
* `"ALPHANUMS(n)"`: An alphanumeric string of length `n`.
* `"EMAIL(n)"`: A valid email address of length `n`.

PROBLEM:
{document}

Output:

9.3 Model Evaluation Prompt

Listing 4: Model Evaluation Prompt

```
You are an **Objective Reasoning Quality Evaluator**. Your task is to rigorously compare the provided "Model Reasoning" against the "Golden Reasoning" based on a three-dimensional scoring rubric.

**--- 1. SCORING RUBRIC (Use ONLY these scores and criteria) ---**
```

```

Rate the Model Reasoning on a scale of 0 to 5 for each dimension.

### A. Logical Validity / Correctness (Weight: High)
* **5:** Perfect. Logically sound, no errors, conclusion is correct. Matches golden logic or is equally sound/correct.
* **4:** Mostly Valid. The reasoning is logically sound and the final answer is correct, but one or two minor, non-critical logical steps are slightly confusing, redundant, or inefficient compared to the golden reasoning.
* **3:** Partially Valid. Minor logical flaw/inefficiency, or correct logic but incorrect final answer (e.g., calculation error).
* **2:** Significantly Flawed. Contains a major logical error early in the process that fundamentally breaks the chain of reasoning, leading to an incorrect final answer.
* **1:** Completely Invalid. Nonsensical or based on incorrect premises.
* **0:** No Reasoning provided.

### B. Coherence / Step-by-Step Flow (Weight: Medium)
* **5:** Perfect Flow. Exceptionally clear, well-structured, easy to follow. Each step builds smoothly.
* **4:** Coherent. The flow is good, but a step might jump slightly ahead or rely on an assumption that isn't explicitly stated, making it slightly less optimal than the golden reasoning.
* **3:** Adequate. Mostly coherent, but the sequence is awkward, or missing minor intermediate steps.
* **2:** Poor Flow. Jumps are frequent, intermediate steps are missing, or the ordering is confusing, significantly disrupting the train of thought.
* **1:** Disjointed. Steps appear randomly ordered or are completely disconnected.
* **0:** No Reasoning provided.

### C. Groundedness / Factuality (Weight: Medium)
* **5:** Perfectly Grounded. All facts and claims are correctly derived from the initial context/prompt. No factual errors.
* **4:** Minor Deviation. The reasoning contains one minor factual error or introduces a trivial, unneeded external fact, but this does not affect the core logical argument.
* **3:** Partially Grounded. Contains one non-critical factual error or uses slightly irrelevant external information.
* **2:** Major Factual Error. Contains a significant factual error or "hallucination" that compromises the reliability of the entire argument.
* **1:** Highly Ungrounded. Reasoning is built primarily on incorrect facts or contradicts the context.
* **0:** No Reasoning provided.

**--- 2. INPUT DATA ---**

**[MUT's Required Reasoning Structure]**
// This defines the expected flow, which the Judge will use to evaluate Coherence and Validity.
The Model was instructed to produce a CHAIN OF THOUGHT with the following four steps:
1. Analyze Business Objectives
2. Analyze Basic Validation (BV)
3. Analyze Business Logic (BL) and Data
4. Establish Test Case Strategy

**[API PROBLEM DOCUMENT]**
{document}

**[GOLDEN REASONING (Ground Truth)]**
{golden_reasoning}

**[MODEL REASONING (To be Evaluated)]**
{model_reasoning}

**--- 3. EVALUATION INSTRUCTION ---**

1. Analyze the [MODEL REASONING] solely in comparison to the [GOLDEN REASONING] and the [PROMPT].
2. Assign a score (0, 1, 2, 3, 4 or 5 ONLY) for each of the three dimensions based on the SCORING RUBRIC.
3. Provide a short, specific justification for **each** score.
4. **You MUST output the final result in the exact JSON format specified below.**
```

```

**--- 4. REQUIRED JSON OUTPUT FORMAT ---**

```json
{
  "Scores": {
    "Logical_Validity": {
      "Score": [The numerical score 0-5],
      "Rationale": "[Brief, specific reason for this score.]"
    },
    "Coherence": {
      "Score": [The numerical score 0-5],
      "Rationale": "[Brief, specific reason for this score.]"
    },
    "Groundedness": {
      "Score": [The numerical score 0-5],
      "Rationale": "[Brief, specific reason for this score.]"
    }
  },
  "Overall_Summary": "[A one-sentence summary of the model's performance compared to the golden reasoning.]"
}
```

END OF INSTRUCTION.
```

References

[1] Treblle, The anatomy of an api in 2023: A comprehensive overview, 2023. URL: https://treblle.com/blog/the-anatomy-of-an-api-in-2023-a-comprehensive-overview.

[2] Imperva, State of api security report 2024, 2024. URL: https://thehackernews.com/2024/03/apis-drive-majority-of-internet-traffic.html.

[3] A. Arcuri, Restful api automated test case generation, ACM Transactions on Software Engineering and Methodology (TOSEM) 28 (2019) 1–37. URL: https://arxiv.org/abs/1901.01538.

[4] T. Le, T. Tran, D.-A. Cao, et al., Kat: Dependency-aware automated api testing with large language models, arXiv preprint arXiv:2407.10227 (2024). URL: https://arxiv.org/abs/2407.10227.

[5] A. Pereira, B. Lima, J. P. Faria, Apitestgenie: Automated api test generation through generative ai, arXiv preprint arXiv:2409.03838 (2024). URL: https://arxiv.org/html/2409.03838v1.

[6] D. Corradini, Z. Montolli, M. Pasqua, M. Ceccato, Deeprest: Automated test case generation for rest apis exploiting deep reinforcement learning, arXiv preprint arXiv:2408.08594 (2024). URL: https://arxiv.org/abs/2408.08594.

[7] Testsigma, Jmeter vs postman | top 10 key differences, 2023. URL: https://testsigma.com/blog/jmeter-vs-postman/.

[8] GeeksforGeeks, Jmeter vs postman for api testing, 2023. URL: https://www.geeksforgeeks.org/software-testing/jmeter-vs-postman-for-api-testing/.

[9] M. Nadăş, L. Dioşan, A. Tomescu, Synthetic data generation using large language models: Advances in text and code, IEEE Access 13 (2025) 134615–134633. URL: http://dx.doi.org/10.1109/ACCESS.2025.3589503. doi:10.1109/access.2025.3589503.

[10] Y. Yu, Y. Zhuang, R. Zhang, Y. Meng, J. Shen, C. Zhang, Regen: Zero-shot text classification

via training data generation with progressive dense retrieval, 2023. URL: https://arxiv.org/abs/2305.10703. arXiv:2305.10703.

[11] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. de Laroussilhe, A. Gesmundo, M. Attariyan, S. Gelly, Parameter-efficient transfer learning for nlp, in: Proceedings of the 36th International Conference on Machine Learning (ICML), volume 97 of *Proceedings of Machine Learning Research*, PMLR, 2019, pp. 2790–2799. URL: http://proceedings.mlr.press/v97/houlsby19a.html, original Adapter paper (often called "Houlsby adapters").

[12] J. Pfeiffer, A. Kamath, A. Rücklé, K. Lee, I. Gurevych, Adapterfusion: Non-destructive task composition for transfer learning, in: Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics (EACL), Association for Computational Linguistics, 2021, pp. 487–503. URL: https://aclanthology.org/2021.eacl-main.39, introduced AdapterHub and the Pfeiffer adapter variant.

[13] G. Gerganov, D. Han, A. Thillobhasker, B. Marie, T. Unsloth, Unsloth: Fast and memory-efficient fine-tuning and inference of large language models, https://github.com/unslothai/unsloth, 2024–2025. URL: https://github.com/unslothai/unsloth.

[14] Hugging Face, Open LLM leaderboard, https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard, 2025. Accessed: 2025-12-04.

[15] LMSYS Org, Chatbot Arena leaderboard with Arena-Hard benchmark, https://lmarena.ai/ and https://huggingface.co/spaces/lmsys/chatbot-arena-leaderboard, 2025. Arena-Hard scores updated December 2025, Accessed: 2025-12-04.

[16] L. Zheng, W.-L. Chiang, Y. Sheng, Z. Li, S. Zhuang, Y. Zhuang, Z. Wu, Z. Zhuang, J. Qin, X. Song, et al., MT-Bench: Multi-turn benchmark for large language models, https://github.com/lm-sys/FastChat/blob/main/fastchat/llm_judge/README.md, 2025. Latest scores as of December 2025.

[17] K. Papineni, S. Roukos, T. Ward, W.-J. Zhu, Bleu: a method for automatic evaluation of machine translation, in: Proceedings of the 40th annual meeting of the Association for Computational Linguistics, 2002, pp. 311–318.

[18] C.-Y. Lin, Rouge: A package for automatic evaluation of summaries, in: Text summarization branches out, 2004, pp. 74–81.

[19] D. Li, B. Jiang, L. Huang, A. Beigi, C. Zhao, Z. Tan, A. Bhattacharjee, Y. Jiang, C. Chen, T. Wu, K. Shu, L. Cheng, H. Liu, From generation to judgment: Opportunities and challenges of llm-as-a-judge, 2025. URL: https://arxiv.org/abs/2411.16594. arXiv:2411.16594.

[20] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, Y. Artzi, Bertscore: Evaluating text generation with bert, 2020. URL: https://arxiv.org/abs/1904.09675. arXiv:1904.09675.

[21] W. Yuan, G. Neubig, P. Liu, Bartscore: Evaluating generated text as text generation, 2021. URL: https://arxiv.org/abs/2106.11520. arXiv:2106.11520.

[22] J. Gu, X. Jiang, Z. Shi, H. Tan, X. Zhai, C. Xu, W. Li, Y. Shen, S. Ma, H. Liu, S. Wang, K. Zhang, Y. Wang, W. Gao, L. Ni, J. Guo, A survey on llm-as-a-judge, 2025. URL: https://arxiv.org/abs/2411.15594. arXiv:2411.15594.