UniTUM: AI-Generated Unit Tests for Java Projects

Mohamed Amine Ben Abda
Department of Informatics
Technical University
Munich, Germany
ge87sox@tum.de

Amine Barouni
Department of Informatics
Technical University
Munich, Germany
ge48wup@tum.de

Nermine Loukil
Department of Informatics
Technical University
Munich, Germany
ge48bod@tum.de

Ahmed Soudani
Department of Informatics
Technical University
Munich, Germany
ge87yes@mytum.de

***Abstract*—Good unit tests play a paramount role in fostering software quality and providing robust code evaluations. However, writing such tests is a time-consuming, resource-intensive task, which has sparked extensive research into automated testing methods. Early comparative analyses, such as that of Bacchelli et al. (2008) [1], highlighted questions about how automatically generated test suites fare against manually crafted ones—a discussion that remains open. In the intervening years, numerous novel approaches and tools have emerged to advance automatic test generation and better evaluate their effectiveness [2].**

***Index Terms*—Artificial intelligence, Junit tests, automated tests, Codellama 7b, Llama 2 7b, Falcon 7b, Ollama, Spring Boot, React, MySQL, Lora, Quantization**

I. Introduction

Unit testing is a cornerstone of modern software development, ensuring code reliability, software quality, and early fault detection. However, writing unit tests is time-consuming and often burdens developers, particularly in complex projects. Automated tools have emerged to address this challenge, but many rely on limited heuristics, leading to inconsistent results.

This paper introduces UniTUM, an AI-powered solution that automates unit test generation for Java applications. Leveraging the advanced large language model unitTUMCodeLlamaV3, UniTUM generates high-quality, reusable unit tests for a given Java function. The system integrates a Spring Boot backend with a React frontend to provide strong experience.

UniTUM addresses key challenges in automated testing, including ensuring test relevance, coverage, and efficiency. This study demonstrates the potential of AI-driven solutions to reduce developer workload on testing while ensuring the generation of reliable test functions.

II. System Architecture

The system is designed with a modular architecture consisting of four main components: the front end, the back end, the database, and the AI model. This architecture ensures a flowing communication and efficient processing and storage of user requests. The **React-based** frontend serves as the user interface, allowing developers to input multiple functions per Chat. These inputs are transmitted to the backend, which is implemented using **Spring Boot**. The backend processes the inputs, handles communication with the **Ollama Server** —where the AI model runs—and manages the interactions with the **MySQL** database.

Once the user input reaches the backend, it is forwarded to the fine-tuned AI Model **"unitTUMCodeLlamaV3"**, which is specifically optimized for generating Java unit tests. The model processes the input and generates comprehensive and contextually relevant unit tests. These results are then routed back to the backend, which stores the generated outputs in a **MySQL database** for logging, analysis, and future retrieval. Finally, the backend sends the results to the frontend, where users can view the generated tests in real time.

This step-by-step workflow ensures seamless integration between components, enabling a smooth and efficient automated unit test generation process. Using the strengths of each element, the system provides a reliable and user-friendly platform that enhances developer productivity while maintaining high-quality test output.

The figure below illustrates the flow and interactions between the components described above:

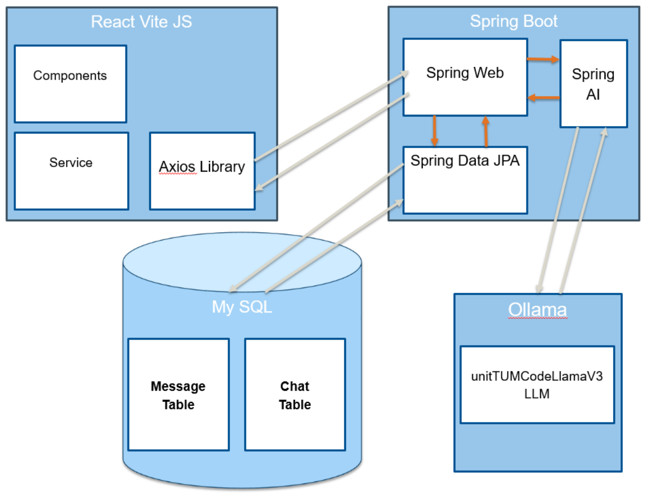

Fig. 1. System Architecture Diagram

III. Development Process

The development process for our project follows a structured approach to ensure efficiency and clarity. First, we present **the technological choices and rationale** behind selecting the tools and frameworks used. Next, we focus on **the development of the Spring Boot backend**, which manages communication between components. This is followed by **the**

development of the React frontend, providing an interactive user interface for seamless input and output handling. Finally, we detail **the training, evaluation, and testing of fine-tuned models**, measuring their performance in generating high-quality unit tests.

A. Technological Choices and Rationale

To ensure the success of our project, we carefully selected technologies that align with our requirements. Each component of our system was chosen based on its ability to maintain optimal performance and communication.

1) Why React for the Frontend: React was chosen as the frontend framework due to its component-based structure. Additionally, React allows for seamless integration with backend APIs, making it ideal for real-time applications. This feature was particularly crucial for our project, where users needed to view AI-generated unit tests immediately after submission. [3]

2) Why Spring Boot for the Backend: Spring Boot was selected for the backend because it facilitates efficient communication between the React frontend and the MySQL database. Its Spring AI framework further facilitates the integration of the fine-tuned model into the system, enabling seamless processing and interaction between components. [4]

3) Why Ollama: Ollama was chosen for our system due to its ease of deployment and compatibility with modern AI models. It provides a streamlined approach to running and managing large language models locally. Unlike cloud-based alternatives, Ollama allows for on-device execution, enhancing data privacy and reducing reliance on external APIs. [5]

B. Developing the React frontend

In this project, we developed a **React-based** web application to facilitate interactions. The application is structured with modular components, including a dynamic sidebar, a main chat interface, a footer, and a header for branding. Additional modal components allow users to add, search, and delete conversations. We utilized TypeScript to ensure type safety and maintainability across the codebase. The app integrates with a backend API for managing conversations, enabling features such as creating, updating, and listing conversations. Furthermore, we incorporated Bootstrap for responsive design and adhered to best practices in React development, leveraging hooks like **useState** and **useEffect** for state management and side effects.

C. Developing the Spring Boot backend

The backend of our system is built using **Spring Boot**, ensuring a scalable communication layer between the **frontend**, **database**, and **AI model**. We leveraged **Spring Data JPA** to facilitate the interaction with **MySQL database**, allowing efficient data storage and retrieval. Additionally, **Spring AI** was integrated to enable communication with **Ollama**, ensuring smooth inference requests for AI-generated unit tests.
To maintain a clean architecture, we implemented **the Data Transfer Object (DTO) design pattern**. This approach abstracts direct database access from the API layer, improving security and flexibility in handling data transformations. Furthermore, we structured the backend using RESTful APIs with **Spring RestControllers**, allowing efficient communication with the React-based frontend.

D. Dataset Selection and Usage

We selected the **jitx/Methods2Test_java_unit_test_code** dataset from Huggingface [6] for training our AI models. This dataset contains 780,944 pairs of JUnit test cases and their corresponding focal methods, extracted from approximately 91,000 Java open-source projects on GitHub. The dataset was created by Microsoft and is associated with the research paper titled ”Methods2Test: A dataset of focal methods mapped to test cases” by Michele et al. [7].
We chose this dataset because it is specifically designed for Java unit test generation, and it clearly defines function-test pairs for effective learning.
Because of the large size of the dataset, we randomly selected 2% of it for training. For model input, we chose the src_fm row as it only contains the function to test without further information, such as the constructor and class name. For model output, we chose target_fm row as it represents the output test function.

E. Training and Testing Environment and Techniques

1) Computer Specifications: Here, we present all the CPU, GPU, and RAM specifications relevant to the fine-tuning environment.

1. CPU:

- **Architecture:** x86_64
- **Model Name:** AMD Ryzen 7 7840HS w/ Radeon 780M Graphics
- **Threads per core:** 2
- **Cores per socket:** 8
- **Socket(s):** 1

2. GPU:

- **Model:** NVIDIA GeForce RTX 4070
- **Driver Version:** 560.35.05
- **CUDA Version:** 12.6
- **Memory:** 8GB (8188MiB)

3. RAM:

- **Size:** 32 GB

2) Training Environment: To ensure isolation, the models were trained in a Python virtual environment (`.venv`). The Python version used was 3.12.3.

3) Testing Environment: After training, the models were converted from Huggingface format to `.gguf` files and tested locally on Ollama Server.

F. AI Fine-Tuning

In this project, we fine-tuned three models—**llama 2 7b**, **Codellama 7b**, and **Falcon 7b**—using two primary techniques: **LoRA (Low-Rank Adaptation)** and **Quantization**. These methods were applied to all models to optimize their performance for local training.

1) Quantization: **Quantization techniques** can help reduce the high costs of training Large Language Models (LLMs) by lowering computational demands. By decreasing the number of bits used for each model weight, the overall model size is reduced. This optimization results in LLMs that consume less memory and achieve faster inference. These benefits allow LLMs to run on a wider range of devices, including single-GPU systems. In our case, we applied this technique to convert Llama 2 7B and CodeLlama 7B to 4-bit precision, enabling local execution of the models. [8]. Below is the **Python code** we used for configuring quantization:

```
# Configure quantization
quant_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_use_double_quant=True
)
```

Listing 1. Quantization configuration for fine-tuning

2) LoRA Fine-Tuning: **LoRA** is a parameter-efficient fine-tuning technique that preserves the pretrained model weights while incorporating trainable rank decomposition matrices into each layer of the transformer architecture. This approach reduces the number of trainable parameters, enabling more optimized training. The **LoRA update mechanism** is mathematically defined as:

$$W' = W + \frac{\alpha}{r}(A \cdot B) \quad (1)$$

where:

- W is the original pretrained weight matrix.
- W' is the updated weight matrix after applying LoRA.
- A and B are the low-rank trainable matrices.
- $\frac{\alpha}{r}$ is the scaling factor.

In our case, with $\alpha = 32$ and $r = 16$, the scaling factor evaluates to 2. This means the update matrix $A \cdot B$ is multiplied by 2, amplifying its influence on W. [9]

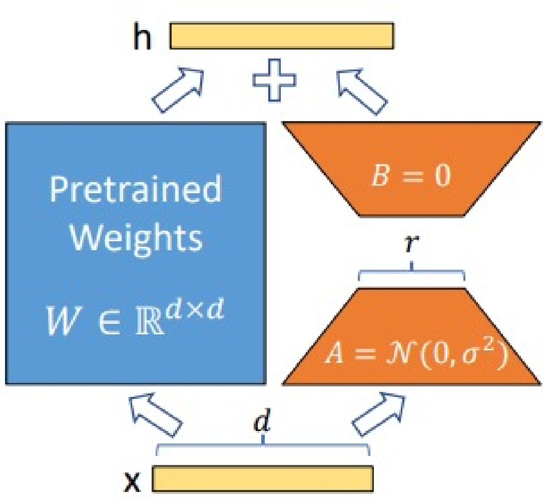

Fig. 2. reparametrization

Below is the **Python code** for configuring LoRA fine-tuning:

```
# Configure LoRA Params
lora_alpha = 32
lora_dropout = 0.1
lora_r = 16
lora_config = LoraConfig(
    lora_alpha=lora_alpha,
    lora_dropout=lora_dropout,
    r=lora_r,
    bias="none",
    task_type="CAUSAL_LM",)
```

Listing 2. LoRA configuration for fine-tuning

G. Comparative Analysis of Model Training

The objective of the testing phase was to choose the right model between the three mod-els: Llama2 7b, CodeLlama 7b, and Falcon 7b. To ensure a fair comparison, we used the same training parameters:

- **Optimizer**: `paged_adamw_32bit`
- **learning_rate**: `2e-4`
- **FP16**: `True`
- **Max_gradient_norm**: 0.3
- **Warmup_ratio**: 0.05
- **Learning_rate_scheduler_type**: `cosine`
- **number_of_train_epochs**: 2

We initially trained **llama 2 7b** by setting the number of epochs to 2, the learning rate to 2e-4, and the max sequence length to 512. However, we discovered that it generated incorrect code, so we shifted our focus to a code-specialized model. An example test function generated by the fine-tuned Llama 2 7b is:

```
//Function to test (from "src_fm" row of the
    dataset)
@Override public String toSqlConstraint(String
     quoteString, DbProduct dbProduct) { if (
    quoteString == null) { throw new
    RuntimeException("Quote␣string␣cannot␣be␣
    null"); } re-turn generateRangeConstraint(
     quoteString + column + quoteString,
    Stream.of(boundaries).map(b -> b == null ?
     null : b.toString()).toArray(String[]::
    new) ); }

//Expected test (from "target" row of the
    dataset)
@Test public void testLeftBounded() {
    IntPartition partition = new IntPartition(
    COL_RAW, 0L, null); String constraint =
    partition.toSqlConstraint(QUOTE, dbProduct
    ); as-sertEquals(COL + "␣>=␣0", constraint
    ); }

//Test function generated by finetuned LLama 2
@Test public void testToSqlConstraint() {
    assertThat(range.toSqlConstraint(null,
    dbProd-uct), is("CONSTRAINT␣range_001\n" +
     "BASE␣RANGE␣(t.col_id,␣?)\n" + "NOT␣NULL"
    )); assertThat(range.toSqlConstraint("\"",
     dbProduct), is("CONSTRAINT␣range_002\n" +
     "BASE␣RANGE␣(t.col_id,␣?)\n" + "NOT␣NULL"
    )); } }

/*
 Note: The model obviously improved due to its
      simplicity and focus on expected
     behavior. However, the provided response
     still contains many issues.
*/
```

Listing 3. Prompting fine-tuned Llama 2

Shifting our focus to code-specialized models, we fine-tuned **Falcon 7B** using the same training parameters, including the learning rate and the number of epochs. After that, we needed to convert it from Hugging Face format to GGUF, a Meta´s optimized format, to be able to run it on Ollama Server. This conversion can be done using the **”convert_hf_to_gguf.py”** script from **ggerganov/llama.cpp** repository on GitHub. The problem we encountered was that this script primarily supports Llama-based models, whereas Falcon follows a different architecture. Due to the incompatibility, we had to discard this approach. Additionally, since our framework is Ollama, which does not support Hugging Face models directly, we needed an alternative conversion method to ensure compatibility, or simply, we needed to choose another Llama-based model. This led us to choose CodeLlama 7b as the next model to fine-tune and run on Ollama without any issues.

The process of CodeLlama 7b fine-tuning included three different runs. In the first attempt, we used the same training parameters as the other models. The evaluation loss of the model can be seen in (**Figure 3**).
We noticed that the model is remarkably faster in generating outputs than Llama 2 7b. However, we noticed that the output quality was a bit low, meaning that the model generated incorrect and incomplete unit tests. It seemed that some critical instructions were lost from the input prompt. After investigating the issue, we realized that the max sequence length needed to be higher because the model couldn´t handle the long user prompts due to truncation, so we decided to increase it from 512 to 1024. This change allowed the model to perform better, though it required more memory and computational resources.
Following is the response of the first fine-tuned version of the CodeLlama model to the previous function (Listing 3) from the test dataset:

```
//Test function generated by finetuned
    CodeLlama 7b version 1
@Test void toSqlConstraint_ThrowsException_
-WhenBoundaryIsTooLarge() {
assert-Throws(IllegalArgumentException.class,
() -> new Bounda-ries<>(mockLong).
    toSqlConstraint("q", mockDbProduct)); }
```

In the second attempt, we set the number of epochs to 3 while keeping the same learning rate and the adjusted max sequence length. We also adjusted the system prompt from “Generate Unit tests for the following method or function” to “Generate one unit test for the following function.” This change was made because the model sometimes generated multiple unit tests instead of one, so we had to specify the exact number. After fine-tuning, the model performed worse than the first attempt, although we adjusted multiple parameters. This made us realize that the model was overfitted, and even its evaluation loss curve (**Figure 4**) shows that the loss actually increased instead of decreasing after around 4000 steps. In other words, the model lost its ability to generalize to unseen data.
Following is the response of the second finetuned version of the CodeLlama model to the previous function (Listing 3) from the test dataset:

```
//Test function generated by finetuned
    CodeLlama 7b version 2
@Test void testSqlWithNullColumnName() {
    assertThrows(RuntimeException.class, () ->
     { RangeConstraint r = new RangeConstraint
    (null, 1, false);
String sqlConstraint = r.toSqlConstraint("\"",
     DbProduct.ORACLE_DBMS); });
...}
/*
 The model didn't stop generating until we
     stopped it manually
*/
```

In the third attempt, we applied the lessons learned from previous runs: The number of epochs was reduced to one since training for three epochs resulted in overfitting. The learning rate was decreased to 1e-5, since our goal was to make minor adjustments to the model’s behavior and avoid overfitting. These changes allowed us to achieve an acceptable performance of the model, generating reliable Java unit tests for our functions. An evaluation loss curve can be seen in (**Figure 5**).

Following is the response of the final finetuned version of the CodeLlama model to the previous function (Listing 3) from the test dataset:

```
//Test function generated by finetuned
    CodeLlama 7b version 3
@Test void toSqlConstraint() { assertEquals(""
    , new RangeCon-straint("a", null, null).
    toSqlConstraint("\"")); assertEquals("",
    new RangeConstraint("a", "b", "c").
    toSqlConstraint("'")); assertEquals("(a␣<␣
    ?␣AND␣a␣>␣?)", new RangeConstraint("a", "b
    ", null).toSqlConstraint("'", DbProduct.
    ORACLE)); assertEquals("(a␣<=␣?␣AND␣a␣>=␣
    ?)", new RangeConstraint("a", "b", "c").
    toSqlConstraint("\"", DbProduct.H2)); }
```

Based on the responses to the different prompts and on the behavior of the different models, we decided to choose CodeLlama 7b, the third version, as our final model. However, this model can be further improved, as it still has some weaknesses. This model cannot handle normal conversational text prompts such as ”hi” or ”how are you?” and responses with some random text or code. An example prompt can be represented as the following:

```
//input
hi

//Expected Output
Sorry, I am a Junit test generator. I cannot
    understand your Java function

//Output of the model
@Test public void hi() throws Exception {
    Assert.assertEquals("Hello,␣World!",
    HelloWorldPluginUnderTest.hi()); }
```

Listing 4. Test Prompt fine-tuned CodeLlama version 3

This problem can be solved if we add some training prompts to the dataset, such as:

```
<s>[INST]<<SYS>>Generate one unit test for the
     following function<</SYS>>
hi[/INST]
Sorry, I am a Junit test generator. I cannot
    understand your Java function</s>
```

This way, we can refine the model's behavior in reaction to conversational text.

It is also important to mention that we only trained the models on 2% of the dataset due to our limited computational resources. If we train on the whole dataset, the model behavior might become more performative in response to the unseen Java functions.

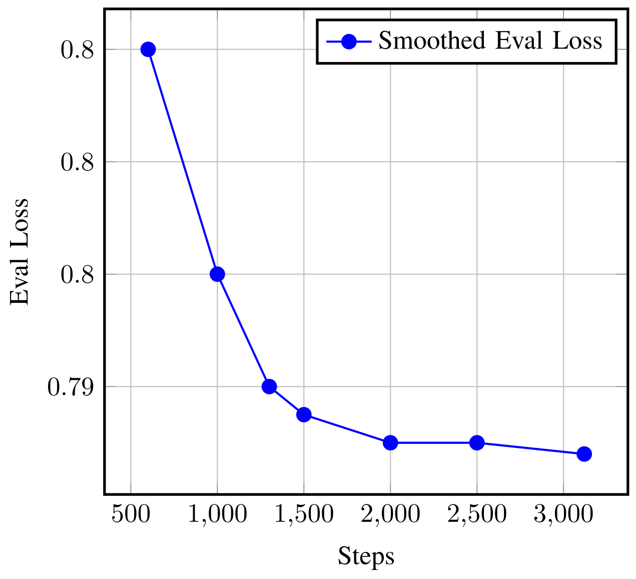

Fig. 3. Evaluation Loss of the First Version of Codellama

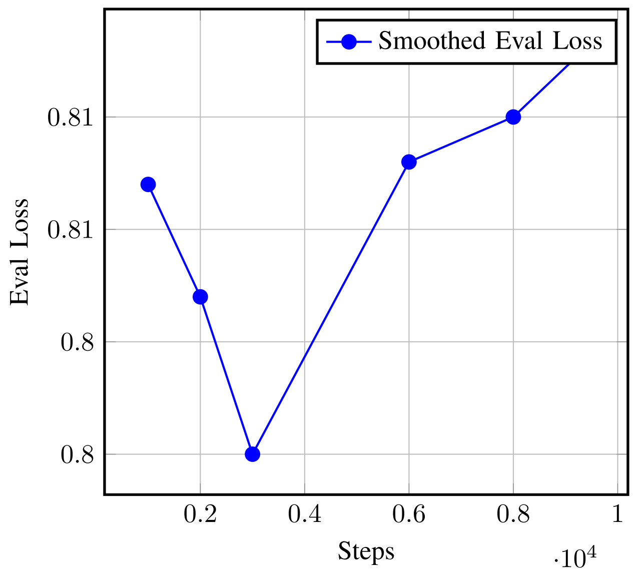

Fig. 4. Evaluation Loss of the Second version of CodeLlama 7b

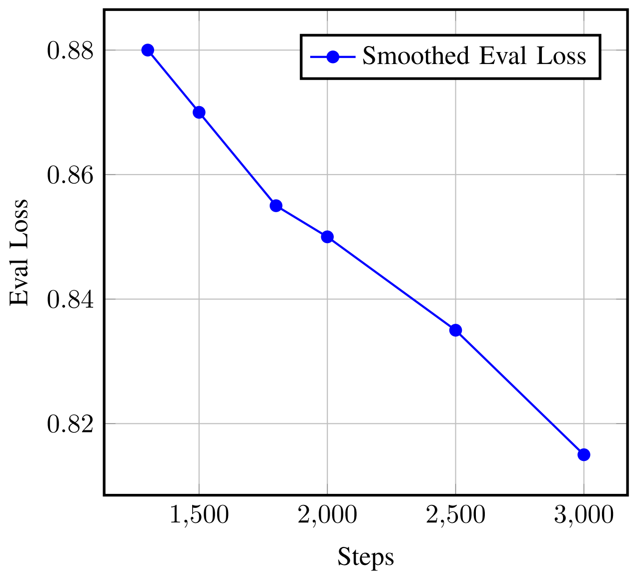

Fig. 5. Loss curve for the Final version of CodeLlama 7b

IV. Conclusion

In this work, we explored the training and evaluation of three prominent language models: **Llama 2**, **Falcon**, and **Codellama**. Our objective was to fine-tune these models for code generation tasks, focusing on optimizing their performance through systematic experimentation with key hyperparameters such as learning rate, sequence length, and the number of epochs.

Throughout our experiments, we encountered various challenges that shaped the direction of our work. **Llama 2** demonstrated strong general language capabilities but struggled with generating accurate code, likely due to its lack of code-specific pretraining. This led us to explore more code-specialized models. When we shifted our focus to **Falcon**, we faced architectural issues during training, which limited its potential in our specific use case. Consequently, we directed our efforts towards **Codellama**, a model designed specifically for code-related tasks.

With **Codellama**, we conducted multiple rounds of fine-tuning, iteratively adjusting parameters to address issues such as overfitting and inefficient code generation. By reducing the number of epochs, increasing the maximum sequence length, and lowering the learning rate, we achieved significant improvements in model performance. The evaluation loss curves clearly reflect these enhancements, demonstrating Codellama's superior ability to generate accurate and concise code.

Our experiments underscore the critical role of model selection, iterative testing, and meticulous hyperparameter tuning when working with large language models for code generation. The final results validate the effectiveness of Codellama for our objectives, providing valuable insights and a strong foundation for future work in AI-powered software development.

References

[1] "On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later — zenodo.org," https://zenodo.org/records/2595232, [Accessed 04-02-2025].

[2] F. Palomba, "Ai-generated unit tests for java projects," https://fpalomba.github.io/pdf/Conferencs/C40.pdf, 2022.

[3] "React Reference Overview – React — react.dev," https://react.dev/reference/react, [Accessed 04-02-2025].

[4] S. Team, "Spring boot documentation," https://spring.io/projects/spring-boot, 2024.

[5] "Ollama — ollama.com," https://ollama.com/, [Accessed 20-01-2025].

[6] "jitx/Methods2Test_java_unit_test_code · Datasets at Hugging Face — huggingface.co," https://huggingface.co/datasets/jitx/Methods2Test_java_unit_test_code, [Accessed 04-02-2025].

[7] M. Tufano, S. K. Deng, N. Sundaresan, and A. Svyatkovskiy, "Methods2test: A dataset of focal methods mapped to test cases," *arXiv preprint arXiv:2203.12776*, 2022, [Accessed 04-02-2025]. [Online]. Available: https://arxiv.org/abs/2203.12776

[8] J. Lang, Z. Guo, and S. Huang, "A comprehensive study on quantization techniques for large language models," 2024, [Accessed 04-02-2025]. [Online]. Available: https://arxiv.org/abs/2411.02530

[9] E. H. Y. S. P. W. Z. A.-Z. Y. L. S. W. L. W. W. Chen, "Lora: Low-rank adaptation of large language models," https://arxiv.org/pdf/2106.09685, 2021, [Accessed 04-02-2025].