electronics

Article

Harnessing Large Language Models for Automated Software Testing: A Leap Towards Scalable Test Case Generation

Shaheer Rehan, Baidaa Al-Bander and Amro Al-Said Ahmad *

School of Computer Science and Mathematics, Keele University, Staffordshire ST5 5BG, UK; y0v95@students.keele.ac.uk (S.R.); b.al-bander@keele.ac.uk (B.A.B.)
* Correspondence: a.m.al-said.ahmad@keele.ac.uk

Abstract: Software testing is critical for ensuring software reliability, with test case generation often being resource-intensive and time-consuming. This study leverages the Llama-2 large language model (LLM) to automate unit test generation for Java focal methods, demonstrating the potential of AI-driven approaches to optimize software testing workflows. Our work leverages focal methods to prioritize critical components of the code to produce more context-sensitive and scalable test cases. The dataset, comprising 25,000 curated records, underwent tokenization and QLoRA quantization to facilitate training. The model was fine-tuned, achieving a training loss of 0.046. These results show the promise of AI-driven test case generation and underscore the feasibility of using fine-tuned LLMs for test case generation, highlighting opportunities for improvement through larger datasets, advanced hyperparameter optimization, and enhanced computational resources. We conducted a human-in-the-loop validation on a subset of unit tests generated by our fined-tuned LLM. This confirms that these tests effectively leverage focal methods, demonstrating the model's capability to generate more contextually accurate unit tests. The work suggests the need to develop novel validation objective metrics specifically tailored for the automation of test cases generated by utilizing large language models. This work establishes a foundation for scalable and efficient software testing solutions driven by artificial intelligence. The data and code are publicly available on GitHub

Keywords: LLM; focal methods; unit testing; test case generation; Llama-2; software testing; QLoRA

check for updates

Academic Editors: Yeong-Seok Seo and Jun-Ho Huh

Received: 25 February 2025
Revised: 21 March 2025
Accepted: 2 April 2025
Published: 4 April 2025

Citation: Rehan, S.; Al-Bander, B.; Al-Said Ahmad, A. Harnessing Large Language Models for Automated Software Testing: A Leap Towards Scalable Test Case Generation. *Electronics* **2025**, *14*, 1463. https://doi.org/10.3390/electronics14071463

1. Introduction

In modern software engineering, effective and comprehensive testing is critical to ensuring the reliability, scalability, and correctness of our everyday applications [1]. This goal has become more challenging due to the complexity of the architectures, dependencies, and codebases in these systems [2]. With software systems growing in this complicated, dynamic, and asynchronous way, served by an enormous number of web and cloud services, manually testing every aspect of software systems has become infeasible, making automation a vital task in the software development lifecycle [3]. Efficient and comprehensive testing that covers user scenarios is critical to detecting bugs early, reducing technical debt, reducing maintenance costs, and ensuring the overall quality of software systems [1]. However, the challenge is to balance test coverage with limited resources and a reduced time to market [3,4].

A key approach to tackle this challenge is the implementation of focal methods [5]. This strategic software testing technique focuses on identifying and prioritizing the complex and critical components of the codebase, ensuring that testing efforts are directed where they will have the greatest impact [6,7]. This approach focuses on highly used, complex, or central functionality components while implementing test scenarios that affect an object's state [8,9]. This allows software engineers to allocate testing resources more effectively, which leads to generating high-quality unit tests [10,11]. This foundational practice ensures that individual methods and functions perform as expected in isolation. However, although combining focal methods and unit tests is beneficial, manual generation for all units, particularly in large-scale systems, can be time-consuming and error-prone [12]. This is where automation is needed to achieve balanced testing coverage, especially with limited resources and a reduced time to market [13].

Although numerous tools and approaches have been developed for automated unit test generation—such as search-based, constraint-based, and random-based techniques [1,14–16]—their effectiveness remains limited [1,17,18]. These methods rely primarily on static code analysis or random input generation, lacking prioritization that considers the importance or complexity of the code. This results in generating many insignificant test cases that miss critical areas where bugs are likely to occur [19]. Furthermore, the inability to grasp the business context of changing software requirements results in incomplete test suites that demand considerable manual effort to maintain and enhance. Thus, many research efforts are underway to explore innovative techniques aimed at enhancing the effectiveness of software testing tasks, with artificial intelligence being among the most promising solutions [1,19].

Artificial intelligence (AI) and its subset, generative AI, have revolutionized numerous fields, including software engineering [20]. Generative AI, particularly in the form of LLMs, has demonstrated remarkable capabilities in understanding and generating human-like text and code. LLMs can automatically generate code and can explain code in natural language, bridging the gap between human understanding and machine execution [21]. These models, trained on vast amounts of data, can capture intricate patterns and relationships within code structures, making them particularly suited for tasks such as code completion, bug detection, and test case generation [22]. LLMs like GPT, PaLM, and Llama have shown impressive results in code-related tasks, often matching or surpassing human performance in certain scenarios [23,24]. The application of these models to software testing has created new possibilities for automating various aspects of software development. These pre-trained models promise to address long-standing challenges in the field, such as the generation of high-quality, context-aware test cases and the automation of test maintenance [25].

Despite the benefits of LLMs being applied in software-engineering-related tasks such as test unit generation [17,26], their adoption and implementation have their own challenges and limitations. The main concern is that LLMs can generate test cases that are syntactically correct (and plausible) but functionally incorrect or not optimized, resulting in a lack of coverage and false results, thus handing the developers a poorly performing test suite [27]. LLMs often have limited understanding of the context and nuances of large software systems, leading to low-quality tests that do not cover domain-specific functionality or edge cases well [28]. Moreover, the dynamic nature of software systems also poses challenges for LLMs in adapting to changes, updating test suites, and maintaining consistency [28]. The black-box nature of LLMs causes explainability and transparency problems as a given test scenario cannot be explained, nor can the completeness of coverage be verified, which are serious problems in safety-critical systems [29]. Additionally, integration with existing software development processes is a challenge since adding LLM-generated tests into continuous integration and continuous deployment (CI/CD) pipelines is not straightforward [29].

Nevertheless, LLMs have the beneficial potential of being able to generate test units in software engineering. To tackle these challenges, addressing the aforementioned issues requires ongoing research, the development of specific tools, and careful thought on how to incorporate LLMs into existing software testing activities. However, as this domain evolves, hybrid methods that combine LLM capabilities with tools and expert human input are likely to be the most effective solution for providing thorough, high-quality software testing. Based on their understanding of code semantics and natural language, LLMs might generate unit tests that cover critical code paths and are sensitive to evolving software requirements, creating a step change in the efficiency and effectiveness of this aspect of the software testing lifecycle.

In this study, we present a novel approach that integrated focal methods with fine-tuning the `Llama-2-7b-chat-hf` model using QLoRA quantization techniques for automated unit test generation. Our experiments, conducted on a curated dataset of 25,000 records, demonstrate the model's capabilities to generate context-aware test cases. Out of three configurations, the chosen fine-tuned model achieved a training loss of 0.046 and showed promising performance in terms of recall (39.5%), an F1 score of 33.95%, and precision of 23.86%, indicating a significant improvement in capturing relevant test scenarios compared to the existing methods. The results were validated in terms of the human-in-the-loop (HITL) process, which also indicates promising results. These results underscore the effectiveness of our approach in enhancing test coverage and reducing redundancy or incorrect test cases, thereby paving the way for scalable and efficient software testing operations.

The remainder of the paper is structured as follows: Section 2 reviews the related work, covering advancements in software testing techniques and test case generation. Section 3 outlines the proposed methodology and describes the experimental setup, including dataset selection, evaluation metrics, and implementation details. Section 4 presents the results and discussion, comparing our method against state-of-the-art approaches and analyzing its performance across different scenarios, considering human-in-the-loop (HITL) validation and threats to validity. Finally, Section 5 summarizes the key findings, highlights the contributions, and discusses potential future directions for improving test case generation.

2. Related Work

Few recent works have investigated different possible ways of leveraging LLM technologies to improve the quality of software engineering by generating test cases related to specific scenarios, detecting bugs, and enhancing the overall software quality assurance activities. With the increasing complexity of software systems, traditional testing methods are often incapable of identifying deep-seated bugs and achieving high test coverage. As a result, researchers have started exploring how LLMs and other AI-assisted tools can be utilized to improve the efficiency and effectiveness of software testing. This section provides an overview of some of the most significant studies that have contributed to this novel field, highlighting high-level advantages, limitations, and future directions for LLM-driven techniques in software testing. These studies show that LLMs integrated into testing workflows can achieve promising results while also identifying challenges to the realization of this potential in broader software development environments.

The work in [30] illustrated how software test case generation validation can be enhanced via the TestGen-LLM tool, which augments existing test classes by generating additional test cases that enable coverage of missed scenarios. The LLMs also filter and evaluate the code, and the evaluation is supplemented by software developers, which helps to improve the efficiency of the model. The results of testing the model showed that 75% of the cases were built correctly, 57% passed reliability checks, and 25% increased test coverage. and, during Meta's test-a-thons, it improved 11.5% of all the classes it was applied to, and 73% of its recommendations were accepted.

The use of LLMs for software testing, specifically in bug detection, was explored in [31], which highlights the limitations of traditional automated test generation tools in identifying complex bugs. It proposes AID, which leverages large language models (LLMs) and differential testing to generate effective test cases and oracles. The paper reports that conventional LLM-based testing achieves only 6.3% precision due to incorrect test oracles. In contrast, compared to state-of-the-art methods, AID improves recall, precision, and F1 scores by up to 1.80×, 2.65×, and 1.66×, respectively. AID's precision reaches up to 85.09%, significantly outperforming other methods that achieved 53.54% in similar tests. The evaluation of datasets TrickyBugs and EvalPlus shows that AID excels in detecting tricky bugs in plausibly correct programs.

The utilisation of ChatGPT (gpt-3.5-turbo) in this field has been cited with considerable success, such as in [32], which explored the potential of using ChatGPT to automate Java unit test generation. The study evaluated the effectiveness of ChatGPT-generated test sets by analyzing key metrics like code coverage, mutation score, and test execution success rate. The experiment used 33 Java programs and generated 33-unit test sets for each, resulting in a total of 1089 tests. The key findings reveal that the best-performing tests, with a 93.5% average code coverage and 58.8% mutation score, were generated with a temperature setting of 0.6. Despite promising results, some generated tests failed to build due to syntax errors, limiting their practical applicability. The paper highlights that ChatGPT-generated tests performed similarly to traditional tools like EvoSuite, although further work is needed to improve reliability and address the limitations.

The paper in [33] proposed a solution for generating test cases using natural language processing models, T5 and GPT-3. The methodology leverages the T5 model for understanding conversational context and the GPT-3 model for refining and generating test cases in natural language. Trained on the WebNLG2020 dataset, the T5 model extracts key phrases from conversations and encodes them, while GPT-3 further improves test case generation performance. The experimental results indicate that this approach can automate test case generation, reducing reliance on human expertise and improving test coverage. The model generated outputs with reasonable performance in several test scenarios, identifying edge cases often missed in manual testing. However, challenges remain, including the computational expense of GPT-3 and inconsistent output in large-scale test case generation. Future work includes improving the model performance and integrating this system with other testing platforms. This study also highlighted the computational costs of this solution.

The study in [34] presented a novel approach to generating test cases for Test-Driven Development (TDD) using a fine-tuned GPT-3.5 model. The model, fine-tuned on a curated dataset of 163,000 method descriptions and test case pairs, achieved superior performance compared to other models. The key results include 78.5% syntactic correctness, 67.09% alignment with requirements, and 61% code coverage, outperforming baselines such as Bloom and CodeT5. An ablation study revealed that fine-tuning improved syntactic correctness by 223%, requirement alignment by 164%, and code coverage by 153%. Furthermore, effective, prompt design increased syntactic correctness by 124%, requirement alignment by 82%, and code coverage by 58%. The study also found that 21.5% of the generated test cases contained errors, with 11.3% being assertion errors, 6.9% syntax errors, and 2.4% value errors.

The authors of [35] presented a new benchmark designed to evaluate large language models' (LLMs) capabilities in generating test cases for software testing. Using 210 Python programs from LeetCode, the benchmark assesses the overall coverage, targeted line/branch coverage, and targeted path coverage. Sixteen LLMs were tested, with GPT-4 achieving the highest overall line and branch coverage at 98.65% and 97.16%, respectively. However, most models struggled with tasks requiring specific program logic comprehension. Notably, only minor improvements were observed in targeted line coverage tasks, with the models improving performance by less than 5% in many cases. The paper highlights the need for advanced reasoning frameworks in LLM-based test case generation to handle complex program paths and logic. The study offers insights into the limitations and future directions for LLMs in software testing.

The paper [36] presented the ChatUniTest framework for unit test generation using large language models. ChatUniTest leverages adaptive focal context and a generation–validation–repair mechanism to generate high-quality unit tests. It aims to address the limitations of traditional program-analysis-based tools and previous LLM-based solutions like TestSpark and EvoSuite. In evaluations across four Java projects, ChatUniTest achieved the highest line coverage compared to TestSpark and EvoSuite. The study found that 40.4% of the generated tests failed due to runtime or syntax errors. A user study further confirmed the value of ChatUniTest, with 89% of the participants reporting its usefulness in generating unit tests, especially for junior developers. However, the authors noted challenges with runtime errors and highlighted future improvements, like supporting more programming languages and enhancing the validation mechanisms. The work in [37] introduced A3Test, a novel deep-learning-based approach for automated test case generation augmented with assertion knowledge and verification of naming consistency. Evaluated on the Defects4j dataset, A3Test significantly outperformed baseline tools like AthenaTest and ChatUniTest. A3Test generated 25.16% to 395.43% more correct test cases, achieved up to 34.29% higher method coverage, and provided 25.64% higher line coverage. In terms of assertions, A3Test generated up to 55.56% more correct assertions compared to other methods. Furthermore, the addition of the assertion component improved the performance by 38%, and verification contributed 23.7%. A3Test was also 20.7% faster than AthenaTest in generating test cases, and the developers found the test cases more readable than those generated by EvoSuite, with 70.51% of the participants in agreement. The recent research mentioned above concerning the integration of LLMs for testing demonstrates notable progress in automating various testing activities, including unit test generation, bug detection, and the enhancement of the software quality assurance process. Studies have utilized advanced LLMs such as GPT, PaLM, and Llama, showing considerable potential in generating code and test cases, thus significantly boosting productivity and effectiveness [30–32]. For instance, TestGen-LLM demonstrates that LLMs could improve test coverage during Meta's software testing events [30]. Similarly, tools like AID [31] and ChatUnitTest [36] have enhanced the precision and recall of generated tests compared to traditional automated methods. However, despite these contributions, several limitations remain. Some work reports that the generated test cases struggle with correctness and context-sensitive tests, resulting in syntactical errors and redundancies [32]. Our work stands apart from the previous research by integrating focal methods with a fine-tuned LLM, specifically the `Llama-2-7b-chat-hf` model, using the QLoRA quantization technique and exploring various parameter-efficient fine-tuning configurations. While prior approaches, such as TestGen-LLM [30], AID [31], and GPT-based test generation [32], primarily focused on generating a large volume of test cases, they often suffered from limited content awareness and high computational costs. In contrast, our approach leverages focal methods to strategically prioritize the critical components of the code, resulting in more context-aware and scalable test case generation. Building on the strengths of LLMs and focal methods, this paper introduces a novel approach that automates unit test generation by combining the structured prioritization of focal methods with the code comprehension and generation capabilities of fine-tuned

LLMs. By fine-tuning LLMs on domain-specific code and test data, our method generates relevant and effective unit tests, specifically targeting the most complex and critical parts of the codebase. This method has the potential to provide a major boost to test coverage, reduce the time and resources needed for exhaustive testing by ensuring that all aspects of functionality are considered, and raise the overall standard of software system quality and reliability.

3. Data and Methodology

This study explores the potential of fine-tuned large language models (LLMs) to automate unit test generation in software testing, aiming to streamline the testing process and enhance its efficiency. The experiment centered on adapting a pre-trained LLM for generating test cases for Java methods by training the model on a specialized dataset of focal methods and their corresponding test cases. Key considerations included handling the complexity of Java syntax, ensuring dataset quality, and mitigating potential biases in the training process. Figure 1 depicts the key phases of the developed method.

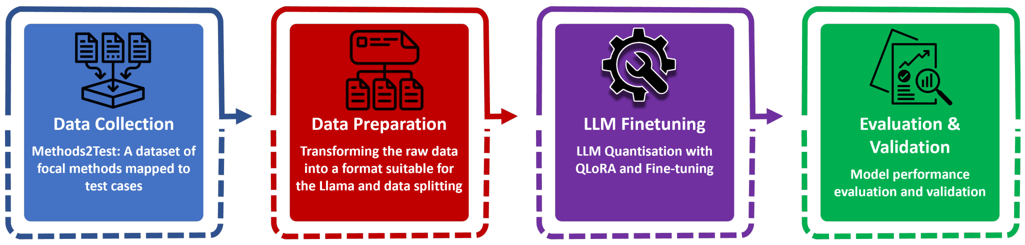
Data Collection
Methods2Test: A dataset of focal methods mapped to test cases
Data Preparation
Transforming the raw data into a format suitable for the Llama and data splitting
LLM Finetuning
LLM Quantisation with QLoRA and Fine-tuning
Evaluation & Validation
Model performance evaluation and validation

Figure 1. Overview of the main phases of the test case generation method.

3.1. Dataset

The publicly available methods2test dataset [5] was selected for its extensive collection of focal methods in Java and their corresponding test cases. The dataset, derived from various Java programming repositories, enables mapping between focal methods and test cases to train the model in identifying patterns for automated test case generation. A focal method, in this context, refers to the specific function or method under test. Additionally, the repository provides metadata that can enhance the learning process by supplying supplementary information [5]. The dataset is organized into a "Corpus" folder containing test cases and corresponding focal methods in formats such as JSON, raw text files, tokenized data, and preprocessed binary data. The text-format corpus was prioritized for its compatibility with the preprocessing workflow. Moreover, the corpus is divided into varying levels of focal context, each incorporating distinct layers of information such as class names, constructor signatures, and public method signatures [5].

The dataset comprises 780,944 focal methods and associated test cases, divided into training and testing. For this study, the `FM_FC` (focal method and focal class name) level of context was chosen as it incorporates the focal method and class name, striking a balance between specificity and simplicity and providing sufficient contextual information to enhance the model's ability to generate test cases without exceeding computational constraints. While deeper levels of context could improve performance, their higher computational demands necessitated reducing the dataset size, which was deemed impractical given the resources available. Using only the focal methods (`FM` level) without additional context was also avoided as it would limit the model's understanding of test case generation patterns. The `FM_FC` level strikes an optimal balance by offering the model sufficient information to generate accurate and task-specific unit tests, avoiding unnecessary complexity or computational overhead [38].

3.2. Data Preprocessing

The dataset containing the `FM_FC` focal context was prepared for the fine-tuning process by transforming the raw data into a format suitable for the `Llama-2-7b-chat-hf` model. The original dataset, comprising `624,022` records, represented focal methods and their corresponding test cases. Given the size constraints imposed by hardware (`A100 GPU`), the dataset was reduced to a manageable size of `25,000` records. This reduction involved incrementally increasing the number of records until a balance between dataset size and available computational resources was achieved. The final size of 25,000 records ensured the feasibility of the fine-tuning process while maintaining a representative sample of the original data. Initially, the raw input and output files were processed and structured, allowing flexible manipulation and preparation.

After that, the raw text data were converted into token `IDs`, the expected input format for the `Llama-2-7b-chat-hf` model. Tokenization was performed using the AutoTokenizer class from Hugging Face's transformer library [39]. To prevent memory spikes, the dataset was tokenized in batches of `5000` records. Each batch was transformed into arrays containing input `IDs`, attention masks, and labels. The token lengths were calculated for each record in the input and output columns, as well as their combined length. These lengths were analyzed to identify and remove outliers that could negatively affect model performance [40]. Entries exceeding `512` tokens for input or `1024` tokens for combined length were excluded, resulting in the removal of `1533` records. This refinement ensured compatibility with the model's context window while minimizing computational overhead.

The filtered dataset was then divided into training and testing subsets using `training 80:testing 20` split. Randomization ensured unbiased splitting, which is critical for evaluating model performance. To standardize record lengths, padding was applied using the UNK token, which avoids conflicts associated with the commonly used EOS token [41]. Attention masks generated during tokenization informed the model of padded versus meaningful tokens, facilitating efficient processing during fine-tuning. The resulting dataset provided a structured, memory-efficient foundation for training the model. This preprocessing pipeline ensured the dataset was optimally prepared for downstream tasks, balancing hardware constraints and model requirements.

3.3. Large Language Model Selection

Selecting an appropriate large language model (LLM) is a critical step in enabling effective fine-tuning for software testing tasks. The chosen model must possess a substantial number of parameters to accommodate training on new data and ensure robust performance in downstream tasks, such as generating unit test cases [42]. Additionally, accessibility and reproducibility are vital; hence, an open-access model that avoids paywalls or subscriptions was prioritized. The `Llama-2-7b-chat-hf` model, republished by NousResearch and available on the Hugging Face platform, was chosen for this study. This model is functionally equivalent to the original `Llama-2-7b`, providing the same architecture and capabilities while being openly accessible. The `Llama-2-7b-chat-hf` model comprises 7 billion parameters and has been pre-trained on 2 trillion tokens sourced from publicly available datasets. Specifically optimized for dialogue-based tasks, this model was provided in the Hugging Face transformer format, facilitating seamless integration into the fine-tuning pipeline. This alignment with the experimental requirements ensured an efficient and reproducible setup for subsequent steps.

It is important to select an LLM that has a large number of parameters to enable it to be trained and fine-tuned on the new data more effectively and to perform well in the required downstream task of test case generation [42]. Another important consideration for the sake of accessibility and repeatability is to select an open-access LLM, i.e., a model that is accessible by users without the need to proceed through any paywalls or subscriptions and the model is openly available to the public. Considering these factors, the `Llama-2-7b-chat-hf` model for NousResearch was selected as the base model. This is the same model as the official Llama-2-7b from Meta, but it has been republished by NousResearch on the Hugging Face platform and is openly accessible, improving the reproducibility of the experiment. The model from NousResearch has been optimized for implementation in dialogue use, and it has been converted into the transformer format from Hugging Face, which will help to streamline the fine-tuning process.

3.4. LLM Parameter-Efficient Fine-Tuning

Parameter-efficient fine-tuning (PEFT) is an efficient way to customize LLMs to a specific task without the computational cost of full fine-tuning. Instead of adjusting the entire model, PEFT fine-tunes only a small set of additional parameters while keeping most of the pre-trained model unchanged. This significantly reduces the time, memory, and storage costs while preserving the model's original knowledge. One of the largest advantages of PEFT is that it avoids catastrophic forgetting, a common issue when fully fine-tuning LLMs, where the model forgets what it already knows. PEFT has also been demonstrated to perform better in low-data regimes and generalize well to new, unseen domains. Many PEFT approaches also improve efficiency via the use of gradient checkpointing, a memory-reduction method that allows models to train without needing to store all intermediate calculations at once.

There are several PEFT methods, each with its own strengths and specialties. Some of the most well-known are adapters, LoRA (Low-Rank Adaptation), QLoRA (Quantized LoRA), prefix-tuning, prompt-tuning, and P-tuning. These approaches enable efficient adaptation of LLMs to various applications with few computational needs. Adapters include small trainable modules within the layers of a frozen pre-trained model. While this strategy reduces memory overhead by reducing the number of model parameters that are to be modified during fine-tuning, it has the disadvantage of requiring the addition of new parameters per task, resulting in increased inference latency and memory usage. LoRA improves computational efficiency and speeds up the training process during fine-tuning by decomposing the weight update matrix into two lower-rank matrices. This reduces the number of trainable parameters and, therefore, the memory requirements. However, the weights of the adaptation layers cannot be quantized in this approach.

QLoRA builds upon LoRA by introducing 4-bit quantization for the weights of the base LLM and the LoRA parameters (adaptation layers) to further reduce memory usage and computational costs without any significant losses in accuracy. The efficiency improvements allow for LLM fine-tuning tasks to be carried out on consumer GPUs, making QLoRA the standout approach for this experiment.

Prefix tuning takes a different approach at reducing memory usage for the fine-tuning of LLMs. This method focuses on optimizing a small set of task-specific vectors that are prefixed to the input embeddings. While this approach boasts a low memory footprint and is efficient when adapting LLMs for simple downstream tasks, the LoRA approaches fare better when the LLM needs to handle complex tasks.

Prompt tuning involves only fine-tuning the continuous input prompt embeddings, which results in fewer parameters being updated. While it can be more efficient in terms of computational resources and memory usage, prompt tuning can struggle to adapt LLMs to complex downstream tasks that require adjustments across several attention layers. Petrov et al. [43] analyzed the effectiveness of context-based PEFT approaches, including prefix tuning and prompt tuning for handling complex downstream tasks, and concluded that these methods, while computationally efficient, struggle to adapt to novel tasks that necessitate new attention structures. Dettmers et al. [44] explained how QLoRA offers the benefit of enabling the fine-tuning of LLMs on consumer hardware with limited memory through the implementation of 4-bit quantization, unlike adapters or LoRA. While prefix and prompt tuning are lightweight methods for parameter-efficient fine-tuning, they often underperform in structured text generation jobs like producing unit test cases for software testing. Fine-tuning deeper attention layers of an LLM helps to produce detailed, structured outputs, which are required when generating software test cases. QLoRA enables fine-tuning the deeper attention layers on standard GPUs by using quantization, which makes it the ideal PEFT method in terms of performance, scalability, and efficiency for fine-tuning the Llama-2-7b model for software test case generation. The object of this paper is to develop an efficient and scalable fine-tuned LLM that can be employed in software test case generation tasks. We have based our choice of QLoRA for parameter-efficient fine-tuning based on an extensive literature review of the candidate methods and the scenarios in which each method is best suited.

PEFT was incorporated to adapt the `Llama-2-7b-chat-hf` model for task-specific requirements without training all parameters. The QLoRA approach introduces a small number of trainable low-rank parameters into targeted layers while freezing the rest, enabling efficient adaptation. The key configurations of the QLoRA method adopted in this study included the following:

- `lora_alpha = 32:` Alpha is the strength of the adapter, and it dictates how much of an impact the adapter has on the model during training. Lower alpha values lessen the weight of the adapter when being merged with the base model, while higher values of alpha inject the adapter more aggressively into the base model. The selected value of 32, while quite high, is generally considered a standard value for alpha for models of this scale as it achieves a balance between strong adaptation of the low-rank fine-tuned layers and retention of the model's existing knowledge [45].
- `lora_dropout = 0.05:` During the training phase of the low-rank layers, a dropout is included to prevent overfitting by randomly deactivating specific connections during the training process. A value of 0.05 indicates that 5% of the neurons would be randomly dropped during the training process to regularize the model. Higher dropout rates can cause the model to underfit, so 5% dropout is an appropriate choice in this case.
- `r = 16:` Sets the rank of low-rank matrices, striking a balance between learning capacity and computational cost. The rank specifies the dimensions of the matrix to be used in the LoRA process. A high rank affords more capacity for the adaptation of learning, but it comes with the cost of higher computational requirements and resource usage. However, if the rank is too low, the adaptation will not be effective, and the model performance will suffer. A rank of 16 was considered appropriate as it allowed for the LoRA process to be executed effectively without increasing the computation overhead to unmanageable levels [45].
- `bias = ‘‘none’’:` Excludes bias parameters to streamline computations. Adding biases in the LoRA process adds additional bias parameters, which are not needed in this case.
- `task_type = ‘‘CAUSAL_LM’’:` Defines the task as causal language modeling, aligning with text generation requirements. Causal LM stands for Casual Language Modeling, and it is the task type used for text generation processes by making predictions of the proceeding words based on the earlier words of the sequence. This is the appropriate choice for this parameter when fine-tuning the Llama-2-7b model for generating software test cases.

- `target_modules = ['up_proj', 'down_proj', 'gate_proj', 'k_proj', 'q_proj', 'v_proj', 'o_proj']`: Focuses adaptation on critical projection layers, including the attention mechanism projection layers ('k_proj', 'q_proj', 'v_proj', 'o_proj') included in the LlamaAttention() structure and the feedforward network (FFN) projection layers ('up_proj', 'down_proj', 'gate_proj') present in the LlamaMLP() structure. The choice regarding these specific target modules is based on the structure of the Llama-2 model. [46].

The QLoRA technique compresses the `Llama-2-7b-chat-hf` to operate efficiently on a single virtual machine. By loading the model in 4-bit precision, memory usage was significantly reduced without compromising accuracy. The following configuration was used to facilitate memory-efficient quantization:

- `load_in_4bit = True`: Ensures that the model is loaded in 4-bit precision instead of the full 32-bit precision. This is the basic concept used in quantization that allows for memory savings.
- `bnb_4bit_compute_dtype = torch.float16`: Defines the data type to be used for computations. Using torch.float16 rather than float32 reduces memory consumption during training, and it helps to maintain adequate precision due to the float16 setting.
- `bnb_4bit_quant_type = "nf4"`: Normalized float 4-bit quantization ensures higher accuracy than integer-based methods, counteracting some of the accuracy losses incurred due to quantization.
- `bnb_4bit_use_double_quant = False`: Avoids double quantization to reduce accuracy loss. Double quantization is used to make the fine-tuning process more memory-efficient at the cost of accuracy, included mainly when weaker hardware is used [39].
- Model: `Llama-2-7b-chat-hf` from the `transformer` library.
- `device_map`: Automatically distributes computation across available GPUs.
- Disabled cache usage and set tensor parallelism to 1.

The `Llama-2-7b-chat-hf` model was loaded, and the `device_map` setting automatically distributed computations across the available GPU, enhancing reproducibility and efficiency. Additional configurations included disabling cache usage to save memory and setting tensor parallelism to 1 due to the single-GPU setup. Finally, `k-bit` (4-bit) training was employed, aligning model layers with low-precision settings to ensure compatibility and resource efficiency. Fine-tuning was conducted and input sequence lengths were capped at 512 tokens to align with the model's context window and hardware constraints. Packing multiple inputs into a sequence was avoided to maintain the integrity of focal methods and their corresponding unit tests. The final dataset included both input and output texts, facilitating effective training of the LLM for test case generation.

Training Hyperparameters

The fine-tuning process was guided by carefully tuned hyperparameters to optimize computational efficiency and model performance. These included the following:

- **Epochs:** 12, to ensure sufficient training cycles.
- **Batch size:** 32 per device for balanced memory usage and gradient updates.
- **Learning rate:** 2×10^{-4}, chosen after iterative testing to prevent overfitting or underfitting.
- **lr_scheduler_type:** "linear", selected because linear decay produced better results than cosine annealing over iterative tests.
- **Gradient clipping:** 0.3, to stabilize gradient updates and improve convergence.
- **Evaluation and checkpointing:** Performed at 500 steps to track progress and save intermediate states.

- **Optimizer:** The `paged_adamw_8bit` optimizer, which reduces memory overhead while maintaining performance.

The learning rate followed a linear scheduler with a warmup ratio of 0.03, gradually increasing to the target rate to stabilize early training stages. Mixed-precision training was enabled to further optimize memory usage and computational speed [39].

3.5. Inference and Model Integration

After training, the trained adapters were merged with the base model to produce a unified fine-tuned model suitable for deployment. Subsequently, the fine-tuned model underwent inference validation using randomly selected test samples from the validation dataset.

3.6. Model Evaluation

The model's performance was rigorously evaluated using the test dataset. Periodic logging of metrics, such as loss and accuracy, provided insights into the model's adaptation and convergence during training. A random sample from the test dataset was used for inference as a proof of concept, ensuring the model was able to communicate via the predefined chat template. This inference step involved generating unit tests for unseen Java methods, demonstrating the model's ability to generalize its learned patterns effectively. Additionally, qualitative assessments of the generated test cases ensured their relevance and syntactic correctness. The evaluation metrics used in this study are explained as follows [47,48]:

- Precision: The proportion of correctly generated responses out of all generated responses.
- Recall: The ratio of correctly generated tokens to the total number of relevant tokens in the validation data.
- F1 Score: The harmonic mean of precision and recall, providing a balanced view of model performance.
- BLEU Score: A metric originally developed for machine translation, used here to evaluate the similarity between generated and reference test cases.

3.7. Data and Code Availability

All data used in this work, including fine-tuned `Llama-2-7b-chat-hf`, scripts for data processing, tokenization, fine-tuning configurations, inference code, and evaluation metric calculations, have been made publicly available for reproducibility purposes via GitHub https://github.com/Shaheer-Rehan/Llama-2-for-Software-Testing (accessed on 21 March 2025).

4. Results and Discussion

This section presents the outcomes of the fine-tuning process, highlighting the key differences in hyperparameter configurations and their impact on model performance. Three configurations with distinct learning rates were evaluated to identify the optimal balance between training and evaluation loss. The discussion emphasizes the final configuration selected for validation.

4.1. Training and Evaluation Loss Metrics

The primary metrics used to assess fine-tuning performance were training loss and evaluation loss. Training loss measures the model's ability to accurately fit the training, and evaluation loss reflects the model's generalization ability on unseen data. Ideally, both metrics should exhibit a steady decline during training. However, in large language models (LLMs), overfitting—evident when training loss decreases but evaluation loss plateaus or increases—can enhance predictive accuracy in specific domains [45]. Loss curves for each configuration are provided below, with solid lines representing training loss and dotted lines representing evaluation loss.

4.1.1. Configuration 1: Learning Rate 3×10^{-5}

The first configuration employed a learning rate of 3×10^{-5} with a cosine annealing scheduler, allowing for a gradual reduction in the learning rate over time. Training was carried out for 12 epochs. The cosine annealing scheduler facilitated a smooth warm-up period, contributing to stable early-stage training, as illustrated in Figure 2.

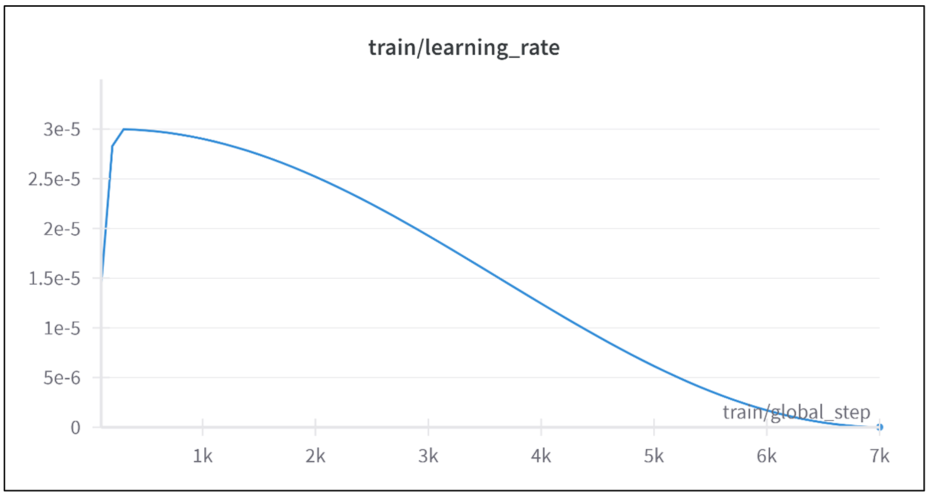

Figure 2. Learning rate schedule during model training, showing a warm-up phase followed by a gradual decay. The learning rate starts at a lower value, increases rapidly during the initial steps (warm-up phase), peaks at approximately 3×10^{-5}, and then gradually decreases as training progresses over approximately 7000 global steps. This approach helps to stabilize training by providing a higher learning rate initially for faster convergence and a lower learning rate later to fine-tune model parameters.

While the loss curves displayed consistent declines, as shown in Figure 3, the evaluation loss plateaued after ∼3500 steps, indicating limited improvement in generalization. The final metrics for this configuration are summarized in Table 1. This configuration exhibited suboptimal performance, as evidenced by the relatively high evaluation loss and limited generalization capability.

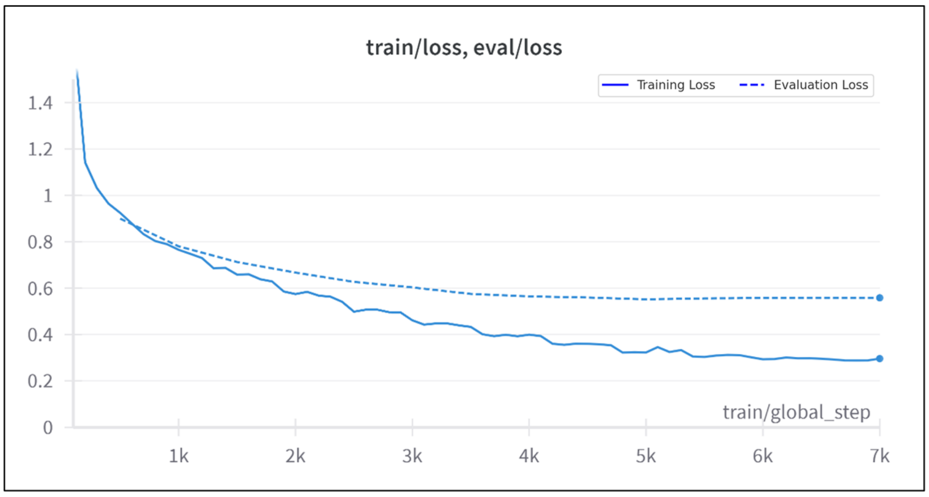

Figure 3. Training and evaluation loss curves for configuration 1 (3×10^{-5}).

Table 1. Evaluation metrics for configuration 1 (3×10^{-5}).

Metric	Value
Training Loss	0.2960
Evaluation Loss	0.5582

4.1.2. Configuration 2: Learning Rate 5×10^{-5}

The second configuration increased the learning rate to 5×10^{-5} and utilized a linear decay scheduler. This configuration achieved better training loss reduction compared to configuration 1, as shown in Figure 4. However, the evaluation loss plateaued after ∼3500 steps, suggesting limited generalization improvement. The final metrics are presented in Table 2. The training was terminated once further reduction in evaluation loss appeared unlikely, ensuring computational efficiency, as illustrated in Figure 5.

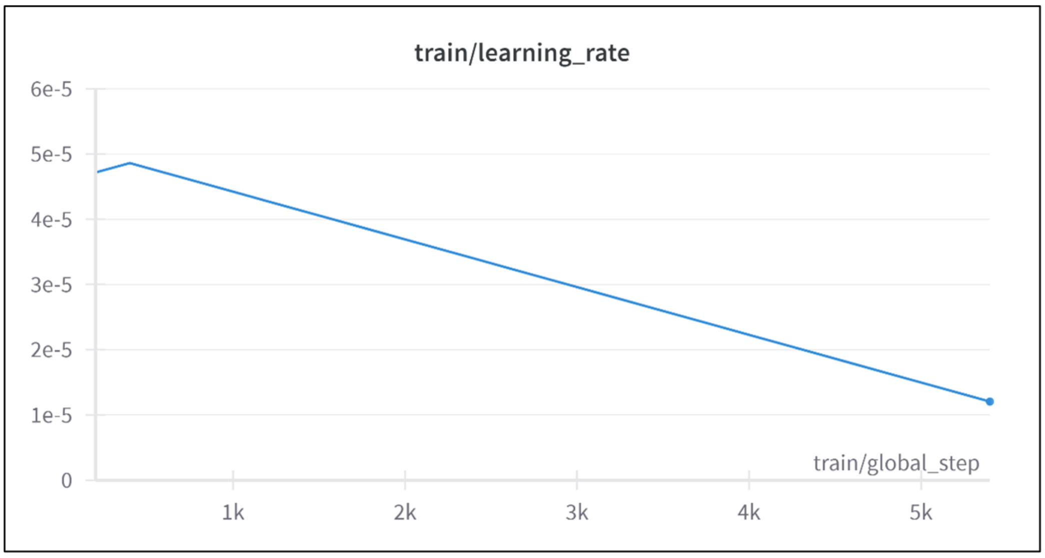

Figure 4. Learning rate decay schedule during model training. The learning rate starts at approximately 5×10^{-5} and decreases linearly over the course of training, reaching a lower value near 1×10^{-5} by the end of approximately 5,000 global steps. This linear decay strategy gradually reduces the learning rate to stabilize learning and fine-tune model performance in later training stages.

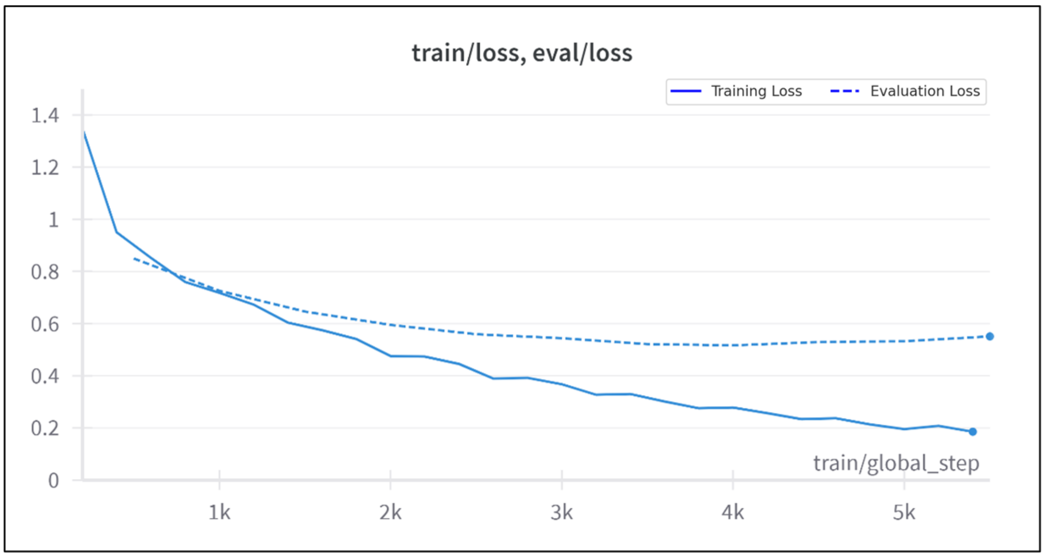

Figure 5. Training and evaluation loss curves for configuration 2 (5×10^{-5}).

Table 2. Evaluation metrics for configuration 2 (5×10^{-5}).

Metric	Value
Training Loss	0.1854
Evaluation Loss	0.5508

4.1.3. Configuration 3: Learning Rate 2×10^{-4}

The third configuration significantly increased the learning rate to 2×10^{-4} while maintaining the linear decay scheduler, as shown in Figure 6. This configuration was selected for the final validation due to its favorable performance trends. While the training loss reached a desirable low value, the evaluation loss increased after ∼4000 steps, confirming overfitting—a phenomenon often advantageous for LLMs in domain-specific tasks [45], as illustrated in Figure 7. The final metrics are detailed in Table 3.

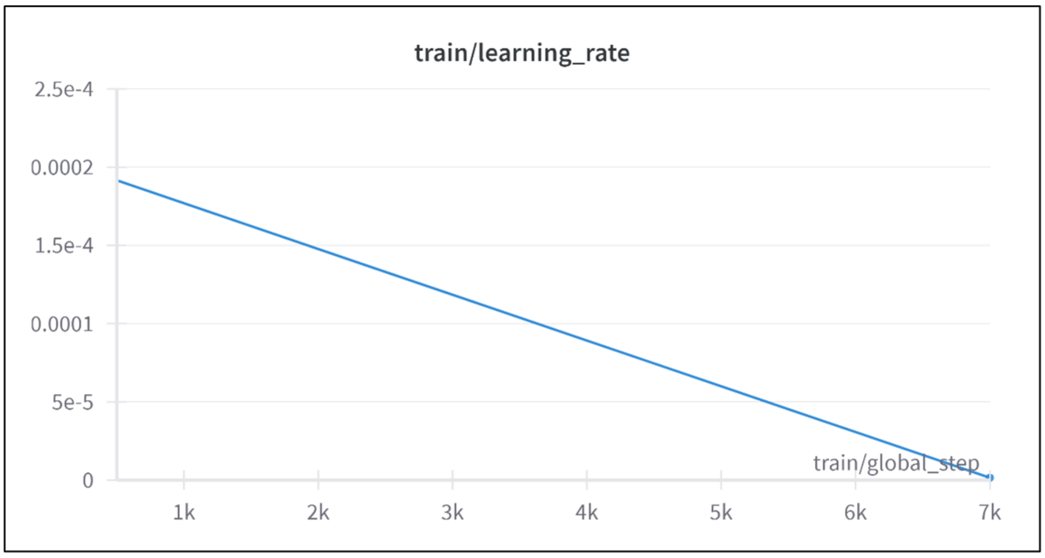

Figure 6. Linear learning rate decay during model training. The learning rate starts at approximately 2×10^{-4} and decreases steadily throughout the training process, reaching close to zero by 7000 global steps. This gradual reduction in learning rate helps to refine model parameters, allowing for more stable convergence and minimizing overshooting during optimization.

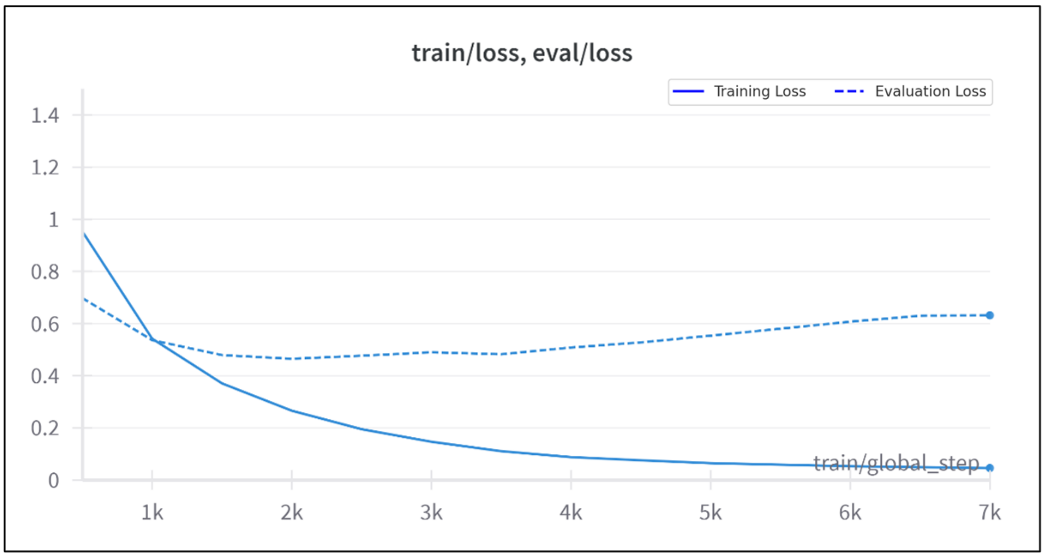

Figure 7. Training and evaluation loss curves for configuration 3 (2×10^{-4}).

Table 3. Evaluation metrics for configuration 3 (2×10^{-4}).

Metric	Value
Training Loss	0.0460
Evaluation Loss	0.6317

This configuration demonstrated strong training performance but limited generalization, attributable to the dataset's small size and computational constraints. The 2×10^{-4} learning rate configuration was ultimately selected for fine-tuning the `Llama-2-7b-chat-hf` model on a software testing dataset. Despite achieving low training loss, the high evaluation loss highlighted challenges in generalization, primarily due to the dataset's limited size (∼25,000 records) and computational constraints (a single 40 GB GPU). Effective fine-tuning of contemporary LLMs often requires larger datasets and greater computational resources to produce models suitable for industrial applications. These findings underscore the importance of aligning computational capacity with model and dataset complexity.

4.2. Validation Using Evaluation Metrics

For this study, where the `Llama-2-7b-chat-hf` model was fine-tuned to generate unit tests for focal methods in Java, validation was conducted by comparing the model's generated outputs with known results (ground truth). Both the baseline and fine-tuned models were loaded, allowing the adapted model weights (QLoRA) to merge with the baseline model. To assess the fine-tuned model, a dataset unseen during training (validation set) was used. This dataset was loaded for preprocessing. The records were tokenized, and entries exceeding a maximum token length of 1024 for inputs or 2048 for combined input–output lengths were filtered out. This step ensures compatibility with computational memory and time constraints. After filtering, 387 records remained. To reduce bias, 300 random samples were selected from the filtered dataset. These samples were converted into lists of inputs (focal methods) and outputs (validation unit tests). Approximately 2400 LLM-generated test cases from our fine-tuned model, along with their corresponding focal methods and validated test cases (hereafter also referred to as reference test cases) from the methods2test dataset, are available on the GitHub page https://github.com/Shaheer-Rehan/Llama-2-for-Software-Testing (accessed on 21 March 2025).

Inference was performed on these samples to generate unit tests for each focal method. The generated tests were considered for further analysis. Tokenization was applied to all the generated tests, enabling compatibility with performance metrics that rely on tokenized data. The evaluation metrics included precision, recall, F1 score, and BLEU score. BLEU score was used to assess the similarity between generated and reference test cases. The combination of standard classification metrics and specialized natural language generation (NLG) metrics is required to comprehensively assess the performance of a fine-tuned LLM for text generation tasks like producing unit test cases from focal methods [49].

Precision is the ratio of correctly generated responses to the model's total generated responses. It provides an indication of how many test cases produced by the LLM are relevant, and high precision scores mean fewer hallucinations [49]. Recall tackles the problem of under-generation in LLMs by assessing how many correct test cases the model was able to produce [49]. Reporting precision and recall when gauging the performance of a fine-tuned LLM in the context of software test generation tasks is important because these metrics help to balance the trade-off between specificity and coverage.

F1 score is a metric that presents a balance of precision and recall and is calculated by determining the harmonic mean of the recall and precision [49]. The F1 score provides a more rounded estimation of the model's actual performance by taking into account recall and precision. Software test cases require high coverage (indicated by recall) as well as high correctness (provided by precision). So, the F1 score serves as a single definitive metric for capturing the trade-off between correctness and coverage.

Finally, BLEU score (Bilingual Evaluation Understudy Score) is an n-gram overlap metric widely used for structured text generation tasks, and it captures the fluency and linguistic quality of the generated responses [49]. In the context of this experiment, the BLEU score indicates how well the generated unit tests from the fine-tuned LLM match the reference unit tests, assessing semantic and syntactic similarities. The BLEU score evaluates the linguistic and structural quality of the responses generated by the LLM, making it an essential metric to gauge the model's performance.

The performance metrics for the fine-tuned `Llama-2-7b-chat-hf` model are summarized in Table 4:

Table 4. Performance metrics for the fine-tuned Llama-2-7b model.

Metric	Value
Precision	23.86%
Recall	39.52%
F1 Score	33.95%
BLEU Score	7.48%

At first glance, the metrics indicate low performance:

- Precision: The score of 23.86% indicates that roughly one-quarter of the model's generated responses are relevant. This result could improve with a larger fine-tuning dataset.
- Recall: The recall of 39.52% suggests that the model captures nearly half of the relevant tokens, which is promising given the small dataset size.
- F1 Score: A score of 33.95% highlights moderate performance, indicating room for improvement.
- BLEU Score: The low score of 7.48% confirms that the generated test cases have limited similarity to the reference test cases, primarily due to dataset constraints. Code–text mismatches, where function implementations do not align perfectly with their corresponding test cases, reduce the model's ability to generate matching outputs. A lack of dataset diversity prevents the model from learning to produce varied yet correct test cases, while ambiguity in gold references means that multiple valid outputs are not captured, leading to lower BLEU evaluations. Additionally, datasets with complex code structures make it harder for the model to generate comprehensive test cases, and noisy annotations from real-world sources introduce errors that further reduce BLEU scores. These constraints collectively hinder the model's ability to produce outputs that align closely with the provided references, resulting in lower BLEU scores.

These results underscore the limitations of the current model, particularly its inability to perform reliably in real-world applications of software test case generation. However, this project demonstrates the potential of fine-tuning large language models for such tasks. Improvements could be achieved with larger datasets, enhanced computational resources, and further fine-tuning. The size of the dataset used to train an LLM during the fine-tuning process is pivotal to the performance of the fine-tuned LLM for a downstream task. With larger datasets, the model has access to more diverse examples of the data, allowing it to learn the more nuanced patterns and adjust better to unseen data. Using smaller datasets can often lead to the model overfitting the training data, resulting in poor generalization.

A study by Mehrafarin et al. [50] demonstrated the impact of the number of fine-tuning samples on the extent of the encoded linguistic knowledge in fine-tuned models.

This study reinforces our conclusion that an increase in dataset size for the fine-tuning process would improve the recoverability of any changes made to the linguistic knowledge base of the model, thereby allowing the model to perform better in generalization tasks on unseen data and exhibit better performance metrics. Ultimately, this project serves as a proof of concept for fine-tuning LLMs for software test case generation, showcasing promising avenues for future research and development in this domain.

4.3. Human-in-the-Loop (HITL) Validation

In software engineering, although the automation of test case generation using AI is essential and can result in a larger number of test cases, it is important to assess the reliability and correctness of the unit test cases generated by large language models (LLMs). Manual validation by test engineers is a commonly used step in both academic and industrial contexts when tests are being generated by LLMs or deep learning techniques that require human validation. Overall, this process will enhance and validate the reliability of systems involving machine learning [51]. In the field of LLMs and unit test generation, the study conducted by Meta using TestGen-LLM [30] clearly shows that, while LLMs are capable of generating numerous unit tests, only a portion of these tests meet the necessary reliability and correctness standards for production use. In this process, engineers are responsible for evaluating and deciding which generated unit test cases to accept or reject.

Therefore, we selected a random sample of six focal methods from the validated test cases in the dataset, https://github.com/Shaheer-Rehan/Llama-2-for-Software-Testing (accessed on 21 March 2025), along with their corresponding LLM-generated unit test cases, totaling fifty-two individual test cases. These fifty-two unit test cases were analyzed from a software engineering perspective, focusing on readability, coverage, object values, edge cases, assertions, and syntax. Table 5 presents the analysis of the test cases generated from the LLM in relation to their focal methods, along with the count of generated test cases using our fine-tuned model. The focal methods listed in the first column of the table are the focal methods published by the publicly available methods2test dataset [5].

Table 5. Software engineering test case analysis.

Focal Method	# Generated Test Cases	Analysis
InstantColumn extends AbstractColumn{InstantColumn, Instant} implements InstantMapFunctions, TemporalFillers{Instant, InstantColumn}, TemporalFilters{Instant}, CategoricalColumn{Instant} { `@Override` boolean isMissing(int rowNumber) { return valueIsMissing(getLongInternal(rowNumber)); } }	15	• Clear and focused assertions, although some minor missing values or assertions. For example, some cover critical rows 0 and 2 for isMissing(0), while some cases did not cover. Redundancy and over-configuration in some cases. • The generated test cases all follow expressive assertions, which makes them easy to understand and follow. • Good coverage. While there are some redundancies, multiple cases capture suitable different behavior settings or input conditions and edge case testing. • Exploration of different configurations. Some tests assert the value for the missing column using setMissingValue (null) or etMissingValue (Instant.now()), which helps to verify that the method behaves correctly under different configurations, ensuring that the missing-value logic is more robust. • Some inconsistency in the expected outcomes as some tests assert row (2) is false, while this should be true.

Table 5. *Cont.*

Focal Method	# Generated Test Cases	Analysis
PackedLocalDateTime extends PackedInstant { static int getSecondOfDay(long packedLocalDateTime) { return PackedLocalTime. getSecondOfDay(time(packedLocalDateTime)); } }	3	• Uses several fixed long values for localDateTime (e.g., 7526268000000L) and expects the output to be a constant value of (12,000 s), which may not be accurate as this is fixed and specific value rather than adapting the range of possible inputs. • There are many repetitions of the same unit test case. • Limited coverage as all of the generated test cases always assert the value of 12,000 for the SecondOfDay. • Adding additional tests for parsing testParse() and testParseWithOffsetRef(), which do not directly assess the focal method (getSecondOfDay). • Some test cases demonstrate correct syntax.
TextColumn extends AbstractStringColumn{TextColumn} { `@Override` Selection isNotIn(String... strings) { Selection results = new BitmapBackedSelection(); results.addRange(0, size()); results.andNot(isIn(strings)); return results; } }	12	• Overall simple unit test cases: The test used a fixed array, such as (““foo””, ““bar””, ““zoo””, ““zoo””) without considering more complex or edge cases or input. • The use of fixed characteristics, such as returning the value 2 for functions getMaxIndex() and getMinIndex(), which does not reflect the method’s true behaviors. • The LLM tests present multiple (closely) identical unit tests with different hard-coded arrays, which does not add any meaningful coverage. • Most of the test cases demonstrate correct syntax, with a focus on core logic for internal boundaries. • The test cases followed structured testing of method behavior, which can help regarding inconsistencies in how duplicates or different input lengths are processed.
Table extends Relation implements Iterable{Row} { static table create() { return new Table(); } } `@Test` void testColumnSizeCheck() { assertThrows(IllegalArgumentException.class, () { double[] a = {3, 4}; double[] b = {3, 4, 5}; Table.create(“test”, DoubleColumn.create(“a”, a), DoubleColumn.create(“b”, b)); });}	7	• Good coverage of detailed assertions on the internal states, such as checking aspects of the table object (such as schema, number of tables, columns, and filters…). • Use of good domain-specific data, such as creating column names with a, b, and c and setting the corresponding expressions like “sum (a.b.c.) > 10”, which simulate realistic scenarios that reflect real use of columns. • Some of the test cases focus on verifying the base behavior of table create () to test additional more complex behaviors that are not part of the focal method. For example, the focal method should focus on creating an empty table. • There are multiple instances of nearly identical unit test cases being repeated. • No-exception testing was conducted as all generated LLM outputs disregarded the column size exception handling.

Table 5. *Cont.*

Focal Method	# Generated Test Cases	Analysis
Table extends Relation implements Iterable<Row> public static table create() return new table();	8	• Good coverage of core functionality; the unit tests clearly verify that the arithmetic operations on columns work correctly. Furthermore, tests create the table with no arguments, resulting in a table with an empty qualified name, a null schema, and empty internal factors, which validate the state of a new table. • Demonstrate verification of extended features as the test cases show not just that the table was created but that, when a group of filters are applied, the internal state of the table is updated accordingly. Also, tests cover scenarios where a schema is set on a table. • The use of precise assertions (e.g., checking table number), which can help to detect deviations from the expected internal state. • Multiple versions of tests for similar functionality, and some showed overly rigid exceptions when relying on hard-coded expected values (e.g., a specific number of columns). • Some tests showed a lack of context on dependencies, where using Table.Creat(relation (“a”, “b”, “c”)) relies on a variable or parameter called relation whose setup is not shown.
DataFrameJoiner public Table leftOuter(Table... tables) return leftOuter(false, tables); @Test public void leftOuterJoinOnAgeMoveInDate() Table table1 = createANIMALHOMES(); Table table2 = createDOUBLEINDEXEDPEOPLE-NameHomeAgeMoveInDate(); Table joined = table1.joinOn(“Age”, “MoveInDate”).leftOuter(true, table2); assertEquals(8, joined.columnCount()); assertEquals(9, joined.rowCount());	7	• Variety of input scenarios as tests covered a good range of conditions (e.g., joining empty tables or single-entry tables). This reflects positively on catching edge cases, such as covering end-of-day tables where included. • Readable and self-contained tests with a focus on table output, focusing on table output to verify the correctness of the functions (merging tables in this case). • Some tests showed limited coverage of real-world scenarios, such as joining multiple large tables (e.g., 2000 rows), but the focus was on an empty or simple table. • Some tests showed redundancies and partial overlaps.

The test cases shared some common strengths but also exhibited similar shortcomings. Across most of the focal methods, the generated tests generally employed clear and expressive assertions that make them easy to read and understand, and some included domain-specific assertions and detailed internal state checks. This aligns with the principles suggested in software testing, where clarity in test cases is crucial for maintainability and fault diagnosis [52]. Some tests incorporated realistic data that simulate real-world usage scenarios, including exception handling assertions, which is beneficial for ensuring that methods perform correctly under practical and useful conditions. Finally, despite redundancies, there is evidence of good coverage for both typical behaviors and edge cases in certain focal methods, which is a promising sign that some of the generated tests capture key functional aspects.

In examining the shortcomings of the generated test cases, many of the tests generated were nearly identical (with approximately 30% of the analyzed set). In a very small number of instances, they were actually the same. Furthermore, approximately 20% of the test cases relied overly on simplistic or hard-coded values, limiting their capability to explore diverse scenarios and edge cases. This lack of diversity in inputs can decrease effectiveness by not exploring a wider range. Moreover, certain test cases where the expected outcomes are inconsistent can lead to confusion in relation to the true method behavior. Finally, some cases overlooked exception handling, leaving a gap in ensuring robustness in error scenarios. Addressing these limitations demands adopting advanced data selection techniques, possibly employing advanced clustering and filtering strategies, to ensure a more representative and diverse training set, ultimately enhancing the model capacity for generating reliable, diverse, and accurate unit tests. Overall, the generated unit tests provide a foundation with clear and expressive assertions, and they simulate realistic scenarios in some cases. However, their dependence on fixed values, the presence of redundant cases, and inconsistent handling of edge cases reveal areas needing improvement. Addressing these issues would result in more comprehensive and robust unit test cases generated by LLMs.

4.4. Validation Discussion

In light of the two validation methods, one is through human-in-the-loop validation for randomized test cases and the other employs automated evaluation methods, such as recall, precision, F1 score, and BLEU score. We found that human-based validation provides advantages over solely depending on automated metrics when assessing the test cases generated by large language models. For example, automated metrics like BLEU focus on surface-level similarity and may assign high scores to test cases that appear syntactically correct but fail to capture critical logical conditions or edge cases. In one instance, a generated test case achieved a high BLEU score but incorrectly validated an edge condition due to a misunderstanding of the input range, which was only identified through human review. Similarly, F1 and recall metrics may overlook semantic errors, such as incorrect assertions or partial coverage of complex input scenarios, which are better detected by human evaluators. These examples highlight the importance of combining automated metrics with human validation to ensure the accuracy, completeness, and functional correctness of generated test cases.

Human oversight offers context-aware judgment that automated metrics often overlook as those metrics were designed for general evaluation purposes. Humans can detect edge cases, redundant tests, and optional biases that an algorithm might overlook. Additionally, human reviewers can access qualitative factors such as readability, maintainability, and the overall reliability of test cases, thus providing a more comprehensive evaluation that focuses on quality. In addition, in unit test cases, there are instances where the syntax of a given generated test case may be missing certain attributes, such as practices or semicolons. Although metrics like the BLUE score could flag these omissions, the test case might still be valid from a context perspective. In such cases, test engineers can make the necessary corrections and consider these test cases for testing. This human intervention can facilitate iterative improvement as this can be fed into the LLM to enhance test case generation over time.

Overall, our findings indicate that integrating human evaluation into the validation of test cases provides considerably more insightful judgment compared to relying solely on automated algorithms. This suggests the need to develop novel validation metrics specifically tailored for the automation of test case generation utilizing large language models or deep learning models. The objective of this development is to reduce the burden placed on human evaluators, whose efforts can be both time-consuming and resource-intensive, thereby enhancing the efficiency of software testing processes that are deployed using LLMs.

4.5. Threats to Validity

In this section, we list potential threats to the validity of our findings following the recommendations [53].

Internal validity: The threats to internal validity lie in potential biases in experiments, such as properties, experimental settings, and metrics used to measure outcomes. To avoid any internal validity threats, especially construct validity, we utilized a supervised dataset that comprises validated test cases and their corresponding focal methods derived from a collection of Java software repositories. Furthermore, we relied on multiple evaluation methods to measure the performance of our output, including precision, recall, F1 score, and BLEU score metrics. In addition, a randomized human-in-the-loop validation was performed for a sample of the test cases. To avoid bias, the third author, an expert in software engineering, independently selected and analyzed a random subset (52 out of approximately 2400 test cases) to ensure an impartial evaluation of test quality. Finally, we conducted fine-tuning of the LLM on three types of configurations to avoid any validity threat from the configuration and hyperparameter perspective. We made these publicly available on GitHub and reported the performance metric comparison of the three configurations.

External validity: This validity type refers to potential threats to the generalizability of the outcome. As we acknowledged this study as a proof of concept, we cannot extend the results to other large language models (LLMs) or programming languages. While the results are promising, further investigations are necessary to draw definitive conclusions.

Reliability: To ensure that our results can be reproduced and that fellow researchers can perform all the experimental settings, we provide access to the fine-tuning code for the `Llama-2-7b-chat-hf` model, which is available online, and all the data generated from our experiments were made available online as well. Furthermore, we relied on a publicly available LLM model with a publicly available validated dataset of unit test cases.

5. Conclusions

This study lays a foundational framework for automating software testing through the fine-tuning of modern large language models (LLMs). The versatility of LLMs enables their application across numerous downstream tasks, including bug localization, test case generation, and test suite optimization. The outcomes of this study highlight the potential of fine-tuned LLMs to revolutionize workflows in software testing, paving the way for more efficient and scalable solutions. The current study provides a proof of concept but leaves ample room for enhancements. With increased computational resources and extended time allocation, several improvements could be pursued:

- Expanded Dataset Size: Increasing the training and validation dataset sizes could significantly enhance the model's performance and generalizability.
- Optimized Hyperparameters: Experimenting with a broader range of learning rates and batch sizes and employing gradient accumulation could lead to an improved hyperparameter configuration.
- Extended Token Lengths: Performing inference with higher maximum token length limits would offer a more comprehensive evaluation of the model's true capabilities.
- Increased Training Epochs: Training the model for additional epochs could lead to more refined and task-specific adaptation.

Based on the results and the human-in-the-loop (HITL) validation, the findings are promising. Some tests demonstrated good coverage of functionalities and addressed multiple scenarios, while certain unit test cases focused on edge cases. However, issues with redundancy, accuracy, and reliability were common, with varying degrees of severity.

Future work should focus on minimizing redundancies and enhancing the accuracy and reliability of the results. This can be achieved by developing more efficient test cases that optimize time and resource allocation during the fine-tuning of large language models for specific datasets. Implementing advanced techniques such as automated data selection, adaptive sampling, and dynamic resource management could streamline the fine-tuning process, ultimately leading to improved model performance and faster deployment in real-world applications, which would contribute to more sustainable practices in model development. This study sets the stage for further advancements in leveraging fine-tuned LLMs for software testing. Future research could explore the following directions:

- Analytical Models for Test Cases: Developing fine-tuned LLMs specifically designed to analyze software test cases, facilitating validation processes.
- Scaled Up Deployment: Creating a robust and accurate fine-tuned model capable of generating software test cases with performance metrics suitable for public deployment.
- Multi-Language Support: Extending the capabilities of fine-tuned models to support software test case generation for various programming languages, such as Python and others.
- New Evaluation Metrics: Develop novel validation metrics specifically tailored for the automation of test cases generated by utilizing large language models.

By addressing these improvement areas and future directions, the potential of LLMs in automating and optimizing software testing can be fully realized. This study underscores the transformative role of artificial intelligence in enhancing efficiency and scalability in software development workflows.

Author Contributions: Methodology, investigation, formal analysis, and writing—original draft, data curation, visualization, S.R.; Methodology, validation, Writing—review and editing, and Writing—original draft, B.A.-B.; Conceptualization, methodology, formal analysis, writing—original draft, validation, project administration, supervision, and Writing—review and editing, A.A.-S.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding. The Article Processing Charge (APC) was fully waived by Electronics (MDPI).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data and code availability: https://github.com/Shaheer-Rehan/Llama-2-for-Software-Testing (21 March 2025).

Acknowledgments: For the purposes of open access, the authors have already granted a CC-BY license over the author-accepted manuscript to Keele University as per this policy.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Wang, J.; Huang, Y.; Chen, C.; Liu, Z.; Wang, S.; Wang, Q. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Trans. Softw. Eng.* **2024**, *50*, 911–936. https://doi.org/10.1109/TSE.2024.3368208.
2. dos Santos, J.; Martins, L.E.G.; de Santiago Júnior, V.A.; Povoa, L.V.; dos Santos, L.B.R. Software requirements testing approaches: A systematic literature review. *Requir. Eng.* **2020**, *25*, 317–337. https://doi.org/10.1007/s00766-019-00325-w.
3. Agh, H.; Azamnouri, A.; Wagner, S. Software product line testing: A systematic literature review. *Empir. Softw. Eng.* **2024**, *29*, 146. https://doi.org/10.1007/s10664-024-10516-x.
4. Souto, S.; D'Amorim, M.; Gheyi, R. Balancing Soundness and Efficiency for Practical Testing of Configurable Systems. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 20–28 May 2017; pp. 632–642. https://doi.org/10.1109/ICSE.2017.64.

5. Tufano, M.; Deng, S.K.; Sundaresan, N.; Svyatkovskiy, A. Methods2Test: A dataset of focal methods mapped to test cases. In Proceedings of the 19th International Conference on Mining Software Repositories, MSR'22, Pittsburgh, PA, USA, 23–24 May 2022; pp. 299–303. https://doi.org/10.1145/3524842.3528009.
6. Pan, R.; Bagherzadeh, M.; Ghaleb, T.A.; Briand, L. Test case selection and prioritization using machine learning: A systematic literature review. *Empir. Softw. Eng.* **2022**, *27*, 29. https://doi.org/10.1007/s10664-021-10066-6.
7. He, Y.; Huang, J.; Yu, H.; Xie, T. An Empirical Study on Focal Methods in Deep-Learning-Based Approaches for Assertion Generation. *Proc. Acm Softw. Eng.* **2024**, *1*, 1750–1771. https://doi.org/10.1145/3660785.
8. Elbaum, S.; Malishevsky, A.G.; Rothermel, G. Prioritizing test cases for regression testing. *Sigsoft Softw. Eng. Notes* **2000**, *25*, 102–112. https://doi.org/10.1145/347636.348910.
9. Elbaum, S.; Malishevsky, A.G.; Rothermel, G. Test Case Prioritization: A Family of Empirical Studies. *IEEE Trans. Softw. Eng.* **2002**, *28*, 159–182. https://doi.org/10.1109/32.988497.
10. Lops, A.; Narducci, F.; Ragone, A.; Trizio, M.; Bartolini, C. A System for Automated Unit Test Generation Using Large Language Models and Assessment of Generated Test Suites. *arXiv* **2024**, arXiv:2408.07846.
11. Yang, L.; Yang, C.; Gao, S.; Wang, W.; Wang, B.; Zhu, Q.; Chu, X.; Zhou, J.; Liang, G.; Wang, Q.; et al. On the Evaluation of Large Language Models in Unit Test Generation. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE'24, Sacramento, CA, USA, 27 October 2024–1 November 2024; pp. 1607–1619. https://doi.org/10.1145/3691620.3695529.
12. Siddiq, M.L.; Da Silva Santos, J.C.; Tanvir, R.H.; Ulfat, N.; Al Rifat, F.; Carvalho Lopes, V. Using Large Language Models to Generate JUnit Tests: An Empirical Study. In Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE'24, Salerno, Italy, 18–21 June 2024; pp. 313–322. https://doi.org/10.1145/3661167.3661216.
13. Dustin, E.; Garrett, T.; Gauf, B. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*; Pearson Education: Upper Saddle River, NJ, USA, 2009.
14. Pacheco, C.; Lahiri, S.K.; Ernst, M.D.; Ball, T. Feedback-Directed Random Test Generation. In Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 20–26 May 2007; pp. 75–84. https://doi.org/10.1109/ICSE.2007.37.
15. Xiao, X.; Li, S.; Xie, T.; Tillmann, N. Characteristic studies of loop problems for structural test generation via symbolic execution. In Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Silicon Valley, CA, USA, 11–15 November 2013; pp. 246–256. https://doi.org/10.1109/ASE.2013.6693084.
16. Harman, M.; McMinn, P. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *IEEE Trans. Softw. Eng.* **2010**, *36*, 226–247. https://doi.org/10.1109/TSE.2009.71.
17. Yuan, Z.; Lou, Y.; Liu, M.; Ding, S.; Wang, K.; Chen, Y.; Peng, X. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. *arXiv* **2024**, arXiv:2305.04207.
18. Fan, A.; Gokkaya, B.; Harman, M.; Lyubarskiy, M.; Sengupta, S.; Yoo, S.; Zhang, J.M. Large Language Models for Software Engineering: Survey and Open Problems. In Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), Melbourne, Australia, 14–20 May 2023; pp. 31–53. https://doi.org/10.1109/ICSE-FoSE59343.2023.00008.
19. Kurian, E.; Briola, D.; Braione, P.; Denaro, G. Automatically generating test cases for safety-critical software via symbolic execution. *J. Syst. Softw.* **2023**, *199*, 111629. https://doi.org/https://doi.org/10.1016/j.jss.2023.111629.
20. Feuerriegel, S.; Hartmann, J.; Janiesch, C.; Zschech, P. Generative AI. *Bus. Inf. Syst. Eng.* **2024**, *66*, 111–126. https://doi.org/10.1007/s12599-023-00834-7.
21. Fui-Hoon Nah, F.; Zheng, R.; Cai, J.; Siau, K.; Chen, L. Generative AI and ChatGPT: Applications, challenges, and AI-human collaboration. *J. Inf. Technol. Case Appl. Res.* **2023**, *25*, 277–304. https://doi.org/10.1080/15228053.2023.2233814.
22. Husein, R.A.; Aburajouh, H.; Catal, C. Large language models for code completion: A systematic literature review. *Comput. Stand. Interfaces* **2024**, *92*, 103917. https://doi.org/10.1016/j.csi.2024.103917.
23. Jiang, J.; Wang, F.; Shen, J.; Kim, S.; Kim, S. A Survey on Large Language Models for Code Generation. *arXiv* **2024**, arXiv:2406.00515.
24. Paul, D.G.; Zhu, H.; Bayley, I. Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review. *arXiv* **2024**, arXiv:2406.00515.
25. Bhatia, S.; Gandhi, T.; Kumar, D.; Jalote, P. Unit test generation using generative ai: A comparative performance analysis of autogeneration tools. In Proceedings of the 1st International Workshop on Large Language Models for Code, Lisbon, Portugal, 20 April 2024; pp. 54–61. https://doi.org/10.1145/3643795.3648396.
26. Schäfer, M.; Nadi, S.; Eghbali, A.; Tip, F. An empirical evaluation of using large language models for automated unit test generation. *IEEE Trans. Softw. Eng.* **2023**, *50*, 85–105. https://doi.org/10.1109/TSE.2023.3334955.
27. Russo, D. Navigating the complexity of generative ai adoption in software engineering. *ACM Trans. Softw. Eng. Methodol.* **2024**, *33*, 1–50. https://doi.org/10.1145/3652154.

28. Jin, H.; Huang, L.; Cai, H.; Yan, J.; Li, B.; Chen, H. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv* **2024**, arXiv:2408.02479.
29. Ozkaya, I.; Carleton, A.; Robert, J.; Schmidt, D. Application of Large Language Models (LLMs) in Software Engineering: Overblown Hype or Disruptive Change? 2023. Available online: https://doi.org/10.58012/6n1p-pw64 (accessed on 30 October 2024).
30. Alshahwan, N.; Chheda, J.; Finogenova, A.; Gokkaya, B.; Harman, M.; Harper, I.; Marginean, A.; Sengupta, S.; Wang, E. Automated Unit Test Improvement using Large Language Models at Meta. In Proceedings of the Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, 15–19 July 2024; pp. 185–196. https://doi.org/10.1145/3663529.3663839.
31. Liu, K.; Liu, Y.; Chen, Z.; Zhang, J.M.; Han, Y.; Ma, Y.; Li, G.; Huang, G. LLM-Powered Test Case Generation for Detecting Tricky Bugs. *arXiv* **2024**, arXiv:2404.10304.
32. Guilherme, V.; Vincenzi, A. An initial investigation of ChatGPT unit test generation capability. In Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing, SAST'23, Campo Grande, Brazil, 25–29 September 2023; pp. 15–24. https://doi.org/10.1145/3624032.3624035.
33. Mathur, A.; Pradhan, S.; Soni, P.; Patel, D.; Regunathan, R. Automated Test Case Generation Using T5 and GPT-3. In Proceedings of the 2023 9th International Conference on Advanced Computing and Communication Systems (ICACCS), Coimbatore, India, 17–18 March 2023; Volume 1, pp. 1986–1992. https://doi.org/10.1109/ICACCS57279.2023.10112971.
34. Alagarsamy, S.; Tantithamthavorn, C.; Arora, C.; Aleti, A. Enhancing Large Language Models for Text-to-Testcase Generation. *arXiv* **2024**, arXiv:2402.11910.
35. Wang, W.; Yang, C.; Wang, Z.; Huang, Y.; Chu, Z.; Song, D.; Zhang, L.; Chen, A.R.; Ma, L. TESTEVAL: Benchmarking Large Language Models for Test Case Generation. *arXiv* **2024**, arXiv:2406.04531.
36. Chen, Y.; Hu, Z.; Zhi, C.; Han, J.; Deng, S.; Yin, J. ChatUniTest: A Framework for LLM-Based Test Generation. In Proceedings of the Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, 15–19 July 2024; pp. 572–576. https://doi.org/10.1145/3663529.3663801.
37. Alagarsamy, S.; Tantithamthavorn, C.; Aleti, A. A3Test: Assertion-Augmented Automated Test case generation. *Inf. Softw. Technol.* **2024**, *176*, 107565. https://doi.org/10.1016/j.infsof.2024.107565.
38. Tufano, M.; Drain, D.; Svyatkovskiy, A.; Deng, S.K.; Sundaresan, N. Unit test case generation with transformers and focal context. *arXiv* **2021**, arXiv:2009.05617.
39. Awan, A.A. Fine-Tuning LLaMA 2: A Step-by-Step Guide to Customizing the Large Language Model. Datacamp, 2023. Available online: https://www.datacamp.com/tutorial/fine-tuning-llama-2 (accessed on 1 July 2024).
40. Zhou, X.; Kim, K.; Xu, B.; Liu, J.; Han, D.; Lo, D. The Devil is in the Tails: How Long-Tailed Code Distributions Impact Large Language Models. In Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), Luxembourg, 11–15 September 2023; pp. 40–52. https://doi.org/10.1109/ASE56229.2023.00157.
41. Marie, B. Padding Large Language Models—Examples with Llama 2. The Kaitchup – AI on a Budget, 2023. Available online: https://kaitchup.substack.com/p/padding-large-language-models (accessed on 21 March 2025).
42. Yu, S.; Fang, C.; Ling, Y.; Wu, C.; Chen, Z. LLM for Test Script Generation and Migration: Challenges, Capabilities, and Opportunities. In Proceedings of the 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS), Chiang Mai, Thailand, 22–26 October 2023; pp. 206–217. https://doi.org/10.1109/QRS60937.2023.00029.
43. Petrov, A.; Torr, P.H.S.; Bibi, A. When Do Prompting and Prefix-Tuning Work? A Theory of Capabilities and Limitations. *arXiv* **2024**, arXiv:2310.19698.
44. Dettmers, T.; Pagnoni, A.; Holtzman, A.; Zettlemoyer, L. QLoRA: Efficient Finetuning of Quantized LLMs. *arXiv* **2024**, arXiv:2305.14314.
45. Labonne, M. Fine-Tuning Your Own Llama 2 Model. Datacamp, 2023. Available online: https://www.datacamp.com/code-along/fine-tuning-your-own-llama-2-model (accessed on 20 December 2024).
46. Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; et al. Transformers: State-of-the-Art Natural Language Processing. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, Online, 16–20 November 2020; Liu, Q., Schlangen, D., Eds.; pp. 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6.
47. Huilgol, P. Precision and Recall in Machine Learning. Analytics Vidhya, 2024. Available online: https://www.analyticsvidhya.com/blog/2020/09/precision-recall-machine-learning/ (accessed on 20 December 2024).
48. Sharma, N. Understanding and Applying F1 Score: AI Evaluation Essentials with Hands-On Coding Example. Arize AI, 2023. Available online: https://arize.com/blog-course/f1-score/ (accessed on 1 July 2024).
49. Jurafsky, D.; Martin, J.H. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*, 3rd ed.; Online manuscript released January 12, 2025; Prentice Hall: Hoboken, NJ, USA, 2025.

50. Mehrafarin, H.; Rajaee, S.; Pilehvar, M.T. On the Importance of Data Size in Probing Fine-tuned Models. In *Findings of the Association for Computational Linguistics: ACL 2022*; Muresan, S., Nakov, P., Villavicencio, A., Eds.; Association for Computational Linguistics: Dublin, Ireland, 2022; pp. 228–238. https://doi.org/10.18653/v1/2022.findings-acl.20.
51. Amershi, S.; Cakmak, M.; Knox, W.B.; Kulesza, T. Power to the People: The Role of Humans in Interactive Machine Learning. *AI Magazine* **2014**, *35*, 105–120. https://doi.org/10.1609/aimag.v35i4.2513.
52. Myers, G.J.; Sandler, C.; Badgett, T. *The Art of Software Testing*; John Wiley & Sons: Hoboken, NJ, USA, 2011.
53. Siegmund, J.; Siegmund, N.; Apel, S. Views on Internal and External Validity in Empirical Software Engineering. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; Volume 1, pp. 9–19. https://doi.org/10.1109/ICSE.2015.24.