# Automatic High-Level Test Case Generation using Large Language Models

***Abstract*—We explored the challenges practitioners face in software testing and proposed automated solutions to address these obstacles. We began with a survey of local software companies and 26 practitioners, revealing that the primary challenge is not writing test scripts but aligning testing efforts with business requirements. Based on these insights, we constructed a use-case → (high-level) test-cases dataset to train/fine-tune models for generating high-level test cases. High-level test cases specify what aspects of the software's functionality need to be tested, along with the expected outcomes. We evaluated large language models, such as GPT-4o, Gemini, LLaMA 3.1 8B, and Mistral 7B, where fine-tuning (the latter two) yields improved performance. A final (human evaluation) survey confirmed the effectiveness of these generated test cases. Our proactive approach strengthens requirement-testing alignment and facilitates early test case generation to streamline development.**

***Index Terms*—Test Case, Use Case, Test Case Generation, Large Language Model**

## I. Introduction

Software testing is not an afterthought but a critical element that must be integrated into the design and development process [14]. It ensures that the software functions as intended [4], meets user requirements [11], reveals bugs [23], provides security [38], and is robust against failures [39]. However, software testing faces significant challenges [17]. Manual testing is time-consuming [5], and as software requirements evolve, maintaining and updating test cases becomes increasingly complex [19].

We conducted a pilot survey by consulting three local software companies to understand their challenges in software testing. We found that the software industry is experiencing a shortage of skilled testers, partly due to testing being perceived as less prestigious than software development. While there is room for improvement in the coding part of testing (i.e., writing test scripts), the main challenge lies in communicating the business requirements with the testers. Given the business requirements (e.g., use cases), most testers suffer more to understand what needs to be tested than converting them to test scripts. Specifically, testers often struggle to effectively identify and design tests for edge and negative cases, which are critical for ensuring the system can handle exceptional and out-of-bound scenarios.

For example, Figure 1 depicts how a simple use case – 'Add Item in Shopping Cart' – can be expanded into a range of test cases that address both expected and exceptional conditions. Testing only the fundamental functionality (in this case, adding an item works as intended) is not enough. The negative test case such as 'Item out of stock' is crucial for ensuring that

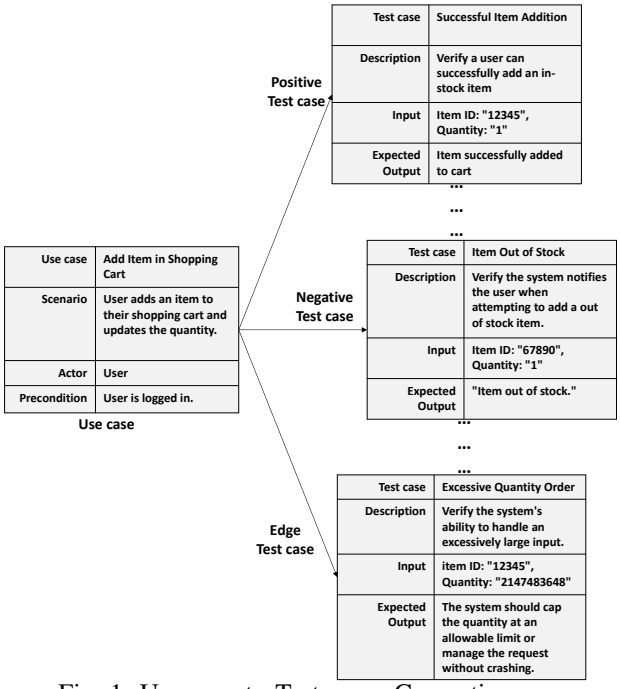


Fig. 1: Use case to Test cases Generation.

the system handles the scenario when an item is not available. On the other hand, edge cases like 'Excessive Quantity Order' examine the system's ability to handle abnormal inputs, such as an excessively large order quantity that could be input by error or as an attempted exploit. If not taken care of, this might be treated unexpectedly by the system. Depending on the implementation and the datatype of the corresponding variable, such overly large numbers can be interpreted as negative values due to integer overflow. This could bypass certain types of logic or validation or may even cause the system to crash.

To that end, we aim to design an automatic system that can generate all possible high-level test cases for a given use case. Unlike low-level test cases (i.e., test scripts), high-level test cases describe what needs to be tested in terms of the functionality and behavior of the software. The goal is to guide the testers during the creation of detailed low-level test script by providing them with a clear understanding of the intended system behavior, including potential negative and edge cases.

We conduct our study in four phases (as summarized in Figure 2). In the initial phase, we conducted a broader survey to verify the findings and assess whether automated high-level test case generation would be a preferable solution for the practitioners. We surveyed 26 software practitioners working

in different roles, companies, and countries. They confirmed that a communication gap indeed exists where ensuring the team understands what needs to be tested (based on business requirements) is a key challenge. Most of them agree that automatic high-level test case generation can be beneficial in alleviating such issues. However, they raised concerns about using external server-based tools due to company policy regarding data security and confidentiality.

In the second phase, we created a dataset of use cases and corresponding test cases to facilitate automatic generation (i.e., training/fine-tuning and evaluating machine learning models). The dataset contains a total of 1067 samples (i.e., 1067 use cases and their corresponding test cases) coming from diverse types of projects (such as E-commerce, Finance, EdTech, Ride-sharing, etc.). The dataset is made publicly available for future study.

In the third phase, we investigate the effectiveness of large language models in generating high-level test cases using our dataset. We evaluated the performance of different large language models (i.e., GPT-4o [1], Gemini [44], LLaMA 3.1 [46] and Mistral [25] ) to generate high-level test cases from use cases. While the pre-trained (raw) models show decent performance with BERTScore as high as 88.43% (F1), fine-tuning smaller models further improves the results. We fine-tuned the LLaMA 3.1 and Mistral (comparatively smaller models) with 80% of our dataset, and their BERTScore improved from 79.85% to 89.94% (for LLaMa) and from 82.86% to 90.14% (for Mistral). These fine-tuned smaller models present a suitable option for organizations that prioritize data confidentiality and require deployment on local servers.

In the final phase, we conducted another survey to assess the quality of the generated test cases by 26 human evaluators. They evaluated the generated test cases in five different aspects (i.e., Readability, Correctness, Completeness, Relevance, and Usability) on a scale from 1 to 5. The overall quality of the automated test cases, generated by the pre-trained (GPT-4o) and fine-tuned (LLaMA 3.1) models, was found to be satisfactory (>3.5 score in every aspect) with strong readability ($avg$ = 4.25), correctness ($avg$ = 4.19), and usability ($avg$ = 4.32).

We propose a proactive strategy focusing on early test case generation immediately following requirement analysis. By generating these test cases early in the development process, we can help mitigate the communication gaps that often lead to misaligned requirements and missed testing scenarios. Thus, developers can identify potential problems earlier and streamline the overall development process.

**Replication Package.** Our code and data are shared at https://anonymous.4open.science/r/uctc

## II. Motivational Survey

In a pilot survey with three local software companies, we found that a communication gap exists between those defining the requirements (like project managers and business analysts) and the teams responsible for testing. This often leads to software that doesn't meet specifications or receives inadequate testing. According to one CTO, "The primary challenge in software testing is not writing test scripts but ensuring the team understands what needs to be tested based on business requirements". This insight led us to explore the potential of high-level test cases – comprehensive outlines of testing needs that could bridge the gap between requirements and testing activities.

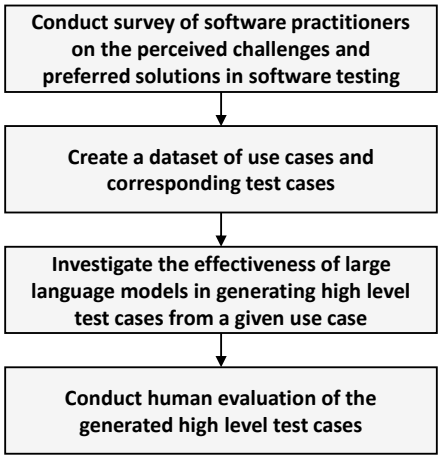


Fig. 2: The four major phases of this study.

To verify these claims, we conducted a broader survey of 26 software practitioners, where we investigated how software development teams perceive and want to address the challenges of aligning their testing efforts with business requirements. We answer the following RQ.

**RQ1**. **How do software development teams perceive their challenges in aligning testing with business requirements, and the role of automated high-level test cases in addressing these challenges?**

We further break down this RQ into the following sub-RQs:

**RQ1.1.** How frequently do software development teams encounter difficulties understanding what needs to be tested?

**RQ1.2.** What is the perceived impact of high-level test cases in guiding testing efforts?

**RQ1.3.** How do software practitioners perceive the automation of high-level test case generation?

**RQ1.4.** What are the practical concerns, such as data confidentiality and organizational policies, that might influence the adoption of a tool for automated high-level test case generation?

### A. Survey Setup

*1) Survey Questions:* We asked each participant 8 questions listed in Table II. The questions focus on exploring the challenges faced by software development teams in aligning their testing efforts with business requirements and evaluating the potential impact of automating high-level test case generation. The survey questionnaire was designed based on our pilot survey, where we discussed the challenges of software testing with industry professionals. The recruitment of those

professionals was conducted mostly by convenient sampling from the authors' professional network.

*2) Survey Participants:* The survey included responses from 26 participants across various roles, organisations, and countries. Table I shows the distribution of the participants over their profession and experience.

TABLE I: Demography of survey participants

| | Years of Experience | | | | |
|---|---|---|---|---|---|
| **Current Role** | **0-2** | **3-5** | **6-10** | **10+** | **Total** |
| Developer | 10 | 5 | 1 | - | 16 |
| Tester | 5 | - | 1 | 1 | 7 |
| Team Lead | - | 2 | - | - | 2 |
| Project Manager | - | 1 | - | - | 1 |
| **Total** | **15** | **8** | **2** | **1** | **26** |

### *B. Challenges in Software Testing (RQ1.1)*

Q1 We asked the participants how often they face difficulties in understanding business requirements when determining what needs to be tested. The responses revealed most practitioners face such issues on a frequent basis.

- ***Always:*** **3.8%**
- ***Often:*** **15.4%**
- ***Sometimes:*** **57.7%**
- ***Rarely:*** **23.1%**
- ***Never:*** **0%**

Q2 We asked the participants about the most challenging aspect of software testing. Most participants (25) reported that the most challenging aspect of software testing is ensuring a clear understanding of what needs to be tested based on business requirements.

- ***Writing test scripts:*** **3.8%**
- ***Determining what to test:*** **96.2%**

### *C. Impact of High-Level Test Cases (RQ1.2)*

Q3 Participants were asked whether they currently use high-level test cases or not. Most participants (22) responded in the affirmative, with a few exceptions (4).

- ***Yes:*** **84.6%**
- ***No:*** **15.4%**

Q4 For the participants who currently use high-level test cases, we further asked them who is primarily responsible for creating those test cases in their organization. While the testers are mostly responsible for writing such test cases, a few exceptions (i.e., project managers, and developers) are also noted.

- ***Developer:*** **19.2%**
- ***Tester:*** **46.2%**
- ***Project Manager:*** **19.2%**
- ***High-level test cases not used:*** **15.4%**

Q5 We asked the participants about the perceived usefulness of high-level test cases in guiding the development of test scripts or manual testing efforts. The responses indicated a generally positive perception of their usefulness.

- ***Very Useful:*** **80.8%**
- ***Moderately Useful:*** **19.2%**
- ***Slightly Useful:*** **0%**
- ***Not Useful:*** **0%**

### *D. Need for Automatic High-level Test Cases (RQ1.3)*

Q6 Participants were asked whether they use any tools for generating or managing high-level test cases. The majority of the participants indicated that they do not use any specific tools for high-level test cases.

- ***Yes:*** **19.2%**
- ***No:*** **80.8%**

Among the tools mentioned, TestRail was the most common, cited by 2 participants. One participant informed that they use a machine-learning based model (developed by themselves) for this task.

- ***TestRail*** **7.7%**
- ***TestCraft*** **3.8%**
- ***Jira*** **3.8%**
- ***ML Model*** **3.8%**

Q7 We inquired about the participants' interest in using a tool that automatically generates high-level test cases based on business requirements. The responses indicated a generally positive attitude towards such a tool.

- ***Yes:*** **57.7%**
- ***Maybe:*** **38.5%**
- ***No:*** **3.8%**

### *E. Practical Concerns of Automatic Tool (RQ1.4)*

Q8 Participants were asked about the existence of any organizational policies that might restrict the use of external server-based tools (for software testing). The majority responded that they have such restrictions due to confidentiality and budget issues.

- ***Yes:*** **Due to Confidentiality: 34.6%, Cost: 11.6%**
- ***Not sure:*** **34.6%**
- ***No:*** **19.2%**

**Summary of RQ1.** Software practitioners often face challenges in understanding business requirements, with 96.2% identifying "determining what to test" as the most difficult aspect of software testing. High-level test cases are widely used, with 84.6% of participants confirming their use, and most participants (80.8%) view them as highly useful in guiding their testing efforts. However, only 19.2% currently use tools for maintaining/generating high-level test cases, despite the majority's willingness to adopt automated solutions if available. Confidentiality and budget concerns are major barriers, as 46.2% of participants reported that their companies restrict the use of tools hosted on external servers.

## III. Automatic Generation of High-Level Test Cases

Our survey revealed that a major challenge in software testing is that testers often struggle with understanding what

TABLE II: Survey questions and their mapping to the Research Questions.

| # | Questions | RQ |
|---|---|---|
| 1 | How often do you or your team face difficulties in fully understanding what needs to be tested? | 1.1 |
| 2 | In your experience, which aspect is more challenging for you or your team for software testing? | 1.1 |
| 3 | Do you or your team currently utilize high-level test cases? | 1.2 |
| 4 | If yes, who is primarily responsible for creating high-level test cases in your organization? | 1.2 |
| 5 | How useful do you believe high-level test cases are in guiding manual testing efforts? | 1.2 |
| 6 | Do you or your team currently use any tool to generate or manage high-level test cases? | 1.3 |
| 7 | Would you be interested in using a tool that automatically generates high-level test cases? | 1.3 |
| 8 | Does your organization currently have policies that restrict the use of external server-based tools? | 1.4 |

needs to be tested rather than just focusing on writing test scripts. In this context, high-level test cases can provide valuable clarity. The survey participants also express interest in automated tools to generate high-level test cases. In this section, we explore the automatic generation of such test cases from business requirements documents. While we could use any form of business requirement document, we focus on use cases because i) they are very common (most software companies maintain them), and ii) they are typically written in a structured format that clearly defines the interactions between users and the system.

**RQ2**. **How effective are large language models in generating high-level test cases?**

**RQ3**. **How effective are smaller models for local server deployment?**

Below, we first describe our dataset creation process. Then, we describe the motivation and approach for each research question and report the results.

### A. Dataset Creation

To the best of our knowledge, no existing dataset provides paired use cases and their corresponding test cases. Hence, we developed a novel dataset leveraging both manually coded and synthesized data to encompass a broad spectrum of real-world and custom scenarios. Table III provides a summary of the dataset's composition, while Table IV outlines the distribution of project types within the dataset.

TABLE III: Summary of dataset

| | Project Type (and Count) | #Usecase | #Testcase |
|---|---|---|---|
| **Human Generated** | Real World Projects (20) | 299 | 1229 |
| | Student Projects (42) | 281 | 961 |
| **Synthesized** | UiPath Projects (106) | 487 | 1416 |
| | **Total** | **1067** | **3606** |

*1) Manual Coding:* The first part of the dataset was created through the collaboration of 300 undergraduate students (120 third-year and 180 fourth-year students) enrolled in advanced software development courses. The students had prior experience in software development through academic projects and internships. The students worked in groups of 5-6 members, with each group supervised by (at least) one faculty member to ensure the quality of the use cases and their corresponding test cases. This effort resulted in a total of 580 use cases and 2190 test cases across two main categories: Real-World Applications and Student Projects.

**Real-world Applications:** In this category, we provided students with 20 diverse real-world applications, covering sectors such as finance, education, ride-sharing, and social media. This approach aimed to capture various practical and complex requirements reflective of industry settings. Thus we collected 299 use cases and 1299 test cases.

TABLE IV: Distribution of project types in our dataset.

| Type | #Student Project | #Real World | #UIPath | Total |
|---|---|---|---|---|
| Vehicle Management | 2 | - | 3 | **5** |
| Travel/Entertinment | 3 | 3 | 2 | **8** |
| Ride-sharing | - | 2 | - | **2** |
| Project Management | 4 | 2 | 14 | **20** |
| Productivity | - | 2 | - | **2** |
| Job Recruitment | - | - | 8 | **8** |
| Healthcare | 4 | - | 13 | **17** |
| Hotel Management | 1 | 1 | - | **2** |
| Finance | - | 2 | 14 | **16** |
| EdTech | 9 | 3 | 4 | **16** |
| E-commerce | 8 | 5 | 17 | **30** |
| Agriculture | 2 | - | 1 | **3** |
| Miscellaneous | 9 | - | 30 | **39** |
| **Total** | **42** | **20** | **106** | **168** |

**Student Projects:** This category comprises 281 use cases and 961 test cases originating from 42 unique term projects. These student projects allowed greater flexibility and creativity that added unique perspectives to the dataset. Each project was supervised by faculty that ensures adherence to academic standards and a coherent structure within the use-case and test-case pairs.

*2) Data Synthesis:* The dataset is further enhanced with synthesized data to facilitate the fine-tuning of smaller models. We utilized UiPath as a source of additional use cases. UiPath is a robotic process automation tool for large-scale end-to-end automation. It is used to automate repetitive tasks without human intervention. We conducted the data synthesis as follows.

**UiPath Projects:** First, we extracted 1237 use cases from 296 UiPath projects. Second, we synthesized test cases for the extracted use cases using GPT-4o (as done in Section III-B). Third, we manually validated 1416 test cases corresponding to 487 use cases (of 106 UiPath projects). Only, the validated samples were incorporated in the original dataset (see Table III) and used for fine-tuning.

*3) Quality Assurance:* We implemented multiple quality control steps throughout the dataset creation process to ensure reliability. While supervisors provided feedback on projects during the creation of use cases and test cases, a final validation check was performed by one author to ensure consistency and correctness. For the synthesized data, we employed the GPT-4o model, which has demonstrated commendable per-

formance in generating test cases from use cases (see Section III-B). We also manually validated all the data points that were incorporated into our dataset.

### *B. High-Level Test Cases with Pre-trained Large Language Models (RQ2)*

*1) Motivation:* Pre-trained LLMs have achieved significant success across various software engineering tasks such as code generation [24], [42], [52], code repair [43], [50], and code documentation [33], [35]. They often perform well without task-specific fine-tuning or specialized training [6]. Hence, in this RQ, we investigate the inherent capability of large language models (LLMs) in generating high-level test cases.

*2) Approach:* In this study, we evaluated GPT-4o [1] and Gemini [44] on generating high-level test cases as detailed below.

**Prompt Engineering.** The interaction with an LLM (e.g., GPT-4) takes place via prompt engineering, where a task description is provided as the input (prompt), and the model performs the desired task (generates test cases) accordingly. There are several ways of prompting i.e., *zero-shot, one-shot, few-shot learning* [6]. In *zero-shot learning*, the model is expected to generate an answer without providing any example. In fact, no additional information other than the task description itself is given in the prompt. On the other hand, one (or few)-shot learning involves giving one (or more than one) example(s) in the prompt to guide the LLMs in the correct direction. In this study, we have developed a simple yet highly effective one-shot prompting approach where we craft a generalized but well-rounded example (use case to test cases pair) that could serve as a proper guideline for the LLM, provide it in the prompt (as an example), and ask the model to generate test cases for another use case by learning from the provided example. The prompt format for generating test cases from a use case is shown in Figure 3. The prompt begins with a single example to demonstrate the use case and test case format. Following this, the specific use case requiring test case generation is provided. Finally, the prompt includes instructions to create test cases that comprehensively cover (i) basic and edge cases, (ii) both positive and negative scenarios, and (iii) valid and invalid inputs.

**Parameter Settings.** There are a number of parameters involved with large language models. One such parameter is *Temperature*, which controls the randomness of the generated output (range 0 to 1). Another randomness parameter is *Top-p* (range 0 to 1), which controls how unlikely words can get removed from the sampling pool by choosing from the smallest possible set (of words) whose cumulative probability exceeds *p*. As recommended by OpenAI official documentation, we set the temperature at a low value (0.0) while keeping top-P at 0.95. We also keep *Frequency* and *Presence penalties* at their default values (0.0), which control the level of word repetition in the generated text by penalizing them based on their existing frequency and presence.

**Evaluation Data.** For evaluation, we randomly selected a subset of 200 human-generated data points from our dataset.

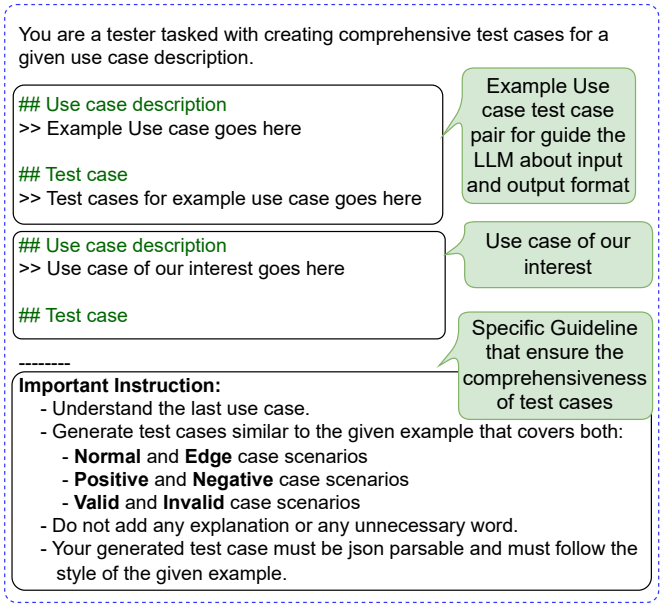


Fig. 3: Prompt for generating test cases from a use case using pre-trained LLMs.

TABLE V: Performance of pre-trained large language models.

| Model | Fine-tuned | Precision | Recall | F1 score |
|---|---|---|---|---|
| Gemini | ✗ | 86.81% | 89.09% | 87.92% |
| GPT-4o | ✗ | **87.63%** | **89.27%** | **88.43%** |

This sample size provides a balance between computational efficiency and statistical validity, enabling a meaningful analysis of the model's ability to generate test cases that align with reference outputs.

**Evaluation Metrics.** Several evaluation metrics exist to compare the machine-generated texts (e.g., automated test cases) with reference texts (e.g., human-generated test cases) such as ROUGE [30], BLEU [37], BERTScore [51]. Among them, we chose BERTScore (using RoBERTa [32] as the underlying embedding model) as our primary evaluation metric. ROUGE (Recall-Oriented Understudy for Gisting Evaluation) measures the overlap between n-grams, longest common subsequences, and word pairs between the generated and reference texts [30]. Conversely, the BLEU (Bilingual Evaluation Understudy) score measures the precision of n-grams between generated and reference text, often focusing on smaller n-grams such as unigrams and bigrams [37]. Hence, they are particularly useful for assessing surface-level or lexical similarity. Their dependence on exact matches can result in low scores even when semantically similar words or phrases are used. Unlike ROUGE and BLEU, BERTScore leverages the pre-trained contextual embeddings from transformer-based models (e.g., BERT, RoBERTa) to measure cosine similarity between the generated and reference texts. Thus, the BERT Score is less sensitive to minor lexical differences, which makes it ideal for our experiment, where the generated test cases should be semantically equivalent to the reference cases, even if they differ in wording.

*3) Result:* The results of the Gemini and GPT-4o models, as presented in Table V, demonstrate that GPT-4o achieves higher scores across all evaluation metrics—Precision, Recall, and F1 Score—compared to Gemini.

1) **Precision:** GPT-4o has a Precision score of 87.63%, showing an improvement over Gemini's 86.81%. This indicates that GPT-4o is more accurate in generating relevant outputs, as a higher Precision suggests fewer irrelevant or erroneous details in the generated test cases compared to Gemini.
2) **Recall:** GPT-4o also outperforms Gemini in Recall, with scores of 89.27% and 89.09%, respectively. This higher Recall implies that GPT-4o retrieves more comprehensive information and is better at including all necessary details in the generated test cases. As Recall reflects the model's ability to capture relevant content similar to the reference cases, GPT-4o's performance suggests a closer alignment with human-generated examples.
3) **F1 Score:** The F1 Score, a harmonic mean of Precision and Recall, further highlights GPT-4o's effectiveness. GPT-4o achieves an F1 Score of 88.43%, an increase from Gemini's 87.92%. The higher F1 Score for GPT-4o underscores its balanced performance in both accuracy and completeness, making it more reliable for generating test cases that closely match human-generated references.

Overall, GPT-4o, with its higher Precision, Recall, and F1 Score, demonstrates superior capability in generating test cases that are both accurate and aligned with reference standards. This suggests that GPT-4o could be more effective for tasks requiring high-quality and semantically relevant outputs, as it consistently performs better than Gemini in all aspects of evaluation.

**Summary of RQ2.** We evaluated two LLMs (i.e., GPT-4o and Gemini) for generating high-level test cases from given use cases. By applying a one-shot prompting strategy, we achieved promising results with GPT-4o, which surpassed Gemini in Precision, Recall, and F1 Score, as evaluated by BERT Score. This result suggests that GPT-4o, aided by prompt engineering, can produce accurate and semantically relevant test cases, making it highly effective for applications that require high-quality, contextually aligned high-level test case generation.

### *C. High-Level Test Cases with Smaller Fine-tuned Models (RQ3)*

*1) Motivation:* Many software companies are hesitant to use third-party LLMs for development tasks due to internal regulations, concerns about confidentiality, and cost. The participants of our survey also raised similar concerns. To that end, we explore the potential of smaller models that come at a cheaper price and can be deployed on the company's local server. Smaller models like LLaMA [46], Mistral [25], Gemma [45], SOLAR [26] can be fine-tuned for a specific task with a quality dataset and can often match the performance of larger general-purpose LLMs [13], [21], [22], [28]. Moreover, these models consume significantly less memory and computational power than larger models, allowing them to be deployed in less powerful machines. Most large-scale models like GPT-4o require a hefty price for continuous usage, whereas an open-source model deployed locally obviates the usage cost. Furthermore, an in-house LLM ensures the privacy and security of confidential data.

*2) Approach:* We fine-tuned the LLaMA 3.1 8B [46] and Mistral 7B [25] to generate quality test cases from a given use case automatically. We chose LLaMA 3.1 8B as it can handle complex language tasks despite its smaller size and fast inference. We also fine-tuned Mistral 7B, as it has been shown to outperform the LLaMA 2 13B model on all benchmarks and even the LLaMA 134B model on some benchmarks [25]. It also offers practical advantages like faster inference time and handling longer sequences at a smaller cost. To assess the impact of fine-tuning, we also evaluated the raw, pre-trained models (before fine-tuning) as a baseline.

**Prompt Engineering.** We used the same prompt format outlined in Section III-B for the fine-tuned models.

**Parameter Settings.** For fine-tuning the models, we utilized QLoRA [13] with 4-bit precision to efficiently reduce memory usage while retaining model performance. We used the Unsloth library, which promises up to 5 times faster fine-tuning with a 70% reduction in memory usage for both LLaMA and Mistral models. For parameter-efficient fine-tuning (PEFT) [21] with LoRA [22], we set $rank = 16$ and $alpha = 16$; this decreases the computational and memory cost of LoRA by storing less information. Such parameter-efficient fine-tuning on high-quality datasets often achieves high performance (comparable to state-of-the-art) while only requiring a fraction of the parameters needed for full fine-tuning [13], [21], [22], [28]. We targeted every linear module to maximize quality. For faster training, dropout and biases were not used. The training was run for 10 epochs with a weight decay of 0.1 and a warm-up ratio of 0.05.

**Evaluation Data.** We used 80% of the data for fine-tuning and the remaining 20% for testing. To be specific, we chose the same 200 data points from Section III-B as our test data (to facilitate a fair performance comparison between the two models). We used the remaining portion (i.e., 867 data points) for fine-tuning the models. While selecting the 200 data points (in Section III-B), we ensured a project-wise split, which means there was no overlap of projects between the fine-tuning and test sets. Thus, the model is evaluated on projects it has not seen during fine-tuning, allowing for a realistic assessment.

**Evaluation Metrics.** Similar to Section III-B, we used the BERTscore to evaluate the generated test cases.

*3) Result:* The results for the fine-tuned models have been reported in Table VI. A significant boost in performance through fine-tuning can be observed. In all three metrics (precision, recall, F1), the fine-tuned models obtain scores in the range of 89-90%, beating all the non-fine-tuned models, including GPT-4o and Gemini (Table V). This further validates our expectation of replicating or surpassing the performance

of huge models like GPT-4o and Gemini using a much smaller model fine-tuned with a quality dataset [13], [21], [22], [28].

TABLE VI: Performance of smaller (fine-tuned) models.

| Model | Fine-tuned | Precision | Recall | F1 score |
|---|---|---|---|---|
| LLaMA 3.1 8B | ✗ | 75.70% | 84.52% | 79.85% |
| LLaMA 3.1 8B | ✓ | **90.11%** | **89.8%** | **89.94%** |
| Mistral 7B | ✗ | 81.36% | 84.44% | 82.86% |
| Mistral 7B | ✓ | **90.21%** | **90.1%** | **90.14%** |

**Summary of RQ3.** Two state-of-the-art open-source models were chosen within the 10B parameter range for fine-tuning, namely LLaMA 3.1 8B and Mistral 7B. We utilized QLoRA with 4-bit precision to fine-tune these models using our limited hardware. Despite the relatively smaller size of these models, they gained significant results after fine-tuning, outperforming massive pre-trained models like GPT-4o and Gemini. These results suggest that software companies can maintain in-house fine-tuned LLMs capable of generating quality test cases while ensuring privacy.

### *D. Human Evaluation*

*1) Motivation:* While automatic metrics provide a standardized way to assess generated test cases, they may lack the nuance and contextual understanding required for a comprehensive evaluation. BERTScore, which works based on semantic similarity, can overlook critical aspects such as logical structure, the presence of edge cases, and specific testing contexts. Two test cases that appear similar in language might differ in the logical sequence of actions or conditions. BERTScore might rate them highly similar, overlooking structural discrepancies that a human evaluator would consider significant. Moreover, BERTscore doesn't inherently capture the adequacy of edge or negative cases in generated test cases. Hence, a generated set can have high similarity to the reference but lacks critical edge or negative scenarios – something a human evaluator would prioritize. Hence, a complete human evaluation of the automated test cases is necessary.

*2) Approach:* We conducted a human evaluation survey to complement the limitations of BERTScore in assessing generated test cases. As mentioned in the earlier sub-sections, we evaluated the automatic models on 200 test samples (i.e., use case and test cases pairs). At 90% confidence level and 10% margin of error, a statistically significant sample size would be 51. Hence, we randomly selected 52 use cases, each with corresponding human-generated, GPT-4o-generated, and LLaMA 3.1-generated test cases. These samples were divided into 13 sets (each containing 4), and distributed to 26 human evaluators (with each set to be reviewed by two evaluators). While there is partial overlap with the previous group of 26 survey participants, the majority are different individuals. To prevent bias, the order of the test cases was randomized, and there was no indication of whether the test cases were human-generated or machine-generated.

The participants were asked to evaluate each set of test cases based on several criteria that are essential to high-quality software testing: Readability (clarity of reading), Correctness (functional accuracy), Completeness (coverage of scenarios), Relevance (focus on essentials), and Usability (ease of transformation into test scripts). Evaluators rated these five aspects on a Likert scale of 1 to 5 (1 being the lowest, 5 being the highest). The questionnaire for the evaluation survey is detailed in Table VII.

We collected responses from all evaluators and calculated the average scores across all five aspects for each sample type (human-generated, GPT-generated, and LLaMA-generated). We measured the average interrater agreement (between the two evaluators of each set) using Cohen's Kappa [10]. To be specific, we used quadratically weighted Cohen's Kappa as it accounts for the ordinal nature of the Likert scale (i.e., 1 to 5) and penalizes larger discrepancies between ratings more heavily than smaller ones [48]. The overall Cohen's Kappa between the two sets of raters was measured to be 0.42, indicating a moderate level of agreement [34].

TABLE VII: Evaluation Criteria and Survey Questions

| Aspect | Question |
|---|---|
| Readability | How clear and easy to understand are the test cases? |
| Correctness | How accurately do the test cases capture the intended functionality of the use case? |
| Completeness | How thoroughly do the test cases cover all relevant scenarios, including positive, edge, and negative cases? |
| Relevance | How well do the test cases focus on essential aspects, avoiding unnecessary or irrelevant steps? |
| Usability | How easily can the test cases be transformed into executable test scripts? |

*3) Result:* The result of the human evaluation is summarized in Table VIII and is discussed below.

**Readability:** Both GPT-4o and LLaMA 3.1 produced clear and easy-to-understand test cases with readability scores of 4.51 and 4.54, respectively. Interestingly, these models even slightly surpassed the readability of human-generated cases, which scored 4.46. This speaks to the inherent language processing capabilities of LLMs, which are pre-trained on extensive datasets to handle a variety of linguistic patterns with fluency.

**Correctness:** Human-generated test cases achieved the highest correctness score of 4.31. GPT-4o followed closely with a score of 4.23, while LLaMA scored lower at 4.15. This implies that GPT-generated test cases are more effective than those produced by LLaMA in capturing core functionalities accurately.

**Completeness:** Human-generated test cases scored the highest for completeness (4.04), while GPT-generated cases had a slightly lower score (3.91), and LLaMA scored the lowest (3.61). It is interesting to see how all the samples, including the human-generated ones, suffer the most in completeness, i.e., covering all possible scenarios including edge cases and

negative cases. The gap is even larger for the automated test cases.

**Relevance:** LLaMA scored marginally higher at 4.02 on relevance, just surpassing GPT's 3.98. Nevertheless, both models fell short of the human-generated score of 4.30. This difference suggests that while machine-generated cases succeed in capturing core functionality, they may occasionally include steps that are irrelevant. LLaMA's slightly better performance (compared to GPT) can be attributed to task-specific fine-tuning.

**Usability:** Human-generated test cases were rated the most usable (4.43), while both GPT-4o and LLaMA followed closely with scores of 4.33 and 4.30, respectively. That suggests that given the high-level test cases, testers can easily convert them to the low-level test cases (i.e., test scripts).

TABLE VIII: Human evaluation of the generated test cases.

| Aspect | Human | GPT-4o | LLaMA 3.1 |
|---|---|---|---|
| Readability | 4.46 | 4.51 | 4.54 |
| Correctness | 4.31 | 4.23 | 4.15 |
| Completeness | 4.04 | 3.91 | 3.61 |
| Relevance | 4.30 | 3.98 | 4.02 |
| Usability | 4.43 | 4.33 | 4.30 |

**Summary of Human Evaluation.** The overall performance of the automated test cases is satisfactory, achieving at least moderate scores (> 3.5) across all evaluation criteria. Both GPT-4o and LLaMA 3.1 test cases demonstrate strong readability, correctness, and usability, though they show limitations in completeness and relevance. A negative correlation is observed between completeness and relevance: GPT-4o compromises dearly on relevance to achieve completeness, often including unnecessary details, while LLaMA prioritizes relevance, which might lead to incomplete scenario coverage. This can be attributed to the inherent design of each model, where GPT's broader training encourages extensive detail, while LLaMA's fine-tuning emphasizes conciseness.

## IV. Discussion

### A. Impact of Enhanced Context

To evaluate whether the LLMs perform better with more context, we assessed the two following approaches: 1) Providing more information about the project in the prompt with a brief overview, and 2) Using RAG to select more relevant examples for one-shot prompting.

*1) Impact of Prompts with Project Descriptions:* Thus far, the LLMs, both pre-trained like GPT-4o and fine-tuned like LLaMA 3.1, were being instructed with one-shot prompts where they only had a use case to generate test cases from. To add more context to our prompts, we enhanced them with a brief description of the project/module/submodule to which the use case in consideration belongs. The improved prompt is shown in Figure 4.

We randomly choose a subset of 100 data points from our previous test dataset of 200. All data points of this subset belong to the real-world project section of our dataset. We

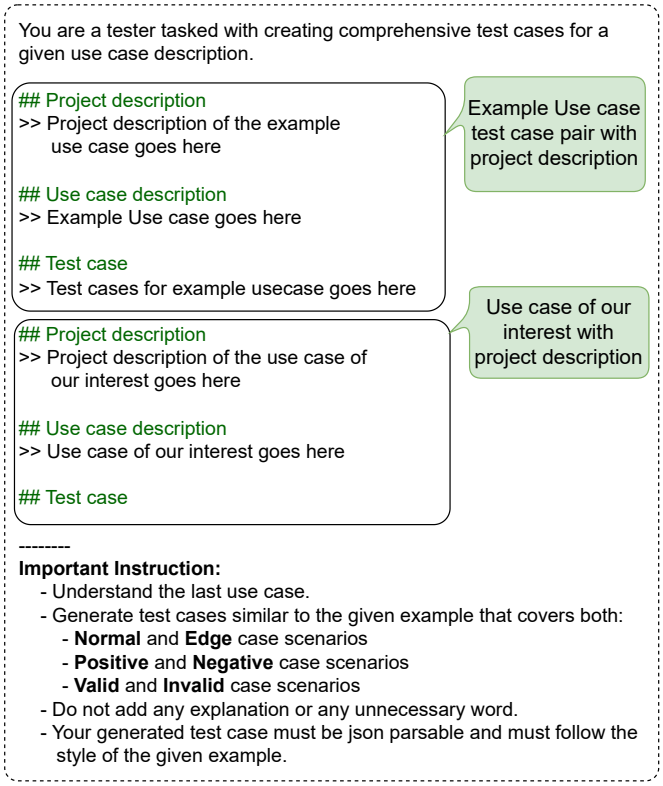


Fig. 4: Prompt for generating test cases from a use case using pre-trained LLMs.

then crafted enhanced prompts for them by inserting a brief overview of their corresponding project/module.

We then evaluated the performance of GPT-4o and the fine-tuned LLaMA 3.1 8B and Mistral 7B models on the subset of the test dataset using the enhanced prompts.

TABLE IX: Performance of models with enhanced prompts using project descriptions.

| Model | Description | Precision | Recall | F1 Score |
|---|---|---|---|---|
| GPT-4o | ✗ | 87.29% | 88.77% | 88.01% |
| GPT-4o | ✓ | 87.08% | 88.87% | 87.95% |
| LLaMA 3.1 8B | ✗ | 89.30% | 89.14% | 89.21% |
| LLaMA 3.1 8B | ✓ | 89.31% | 89.23% | 89.26% |
| Mistral 7B | ✗ | 89.40% | 89.34% | 89.35% |
| Mistral 7B | ✓ | 89.49% | 89.14% | 89.31% |

From Table IX, it is evident that the enhanced prompts do not make any significant difference in the performance of the models. This could be attributed to two possible reasons: 1) the use cases written by our human annotators are already self-sufficient, and the addition of a project description does not contribute much to the context, and 2) LLaMA 3.1 8B and Mistral 7B have already been fine-tuned without the enhanced prompts; as a result, they might not require the additional context to generate quality test cases.

*2) Impact of Using RAG:* Studies find that relevant examples given in the prompt can improve LLM performance [16], [24]. Hence, we integrated Retrieval Augmented Generation (RAG) [29] utilizing Gemini Embedding and Chroma-DB

as our document store. Our knowledge base consists of use case–test case pairs from the student project section of our human-generated dataset. Through RAG, we retrieve the most relevant use cases from this knowledge base, which we then provide as examples in our one-shot learning prompt.

We examined the impact of incorporating RAG on the same subset of 100 data points as used in Section IV-A1, with results summarized in Table X. Interestingly, performance gains were not as substantial as anticipated. This is because our use case–test case pairs are mostly distinct, as they come from different projects. Consequently, the retrieved examples may not be as relevant, offering limited support to the LLM and resulting in nearly equivalent performance.

TABLE X: Performance of models with enhanced prompts using RAG.

| Model | RAG | Precision | Recall | F1 Score |
|---|---|---|---|---|
| GPT-4o | ✗ | 87.29% | 88.77% | 88.01% |
| GPT-4o | ✓ | 88.22% | 88.78% | 88.49% |
| LLaMA 3.1 8B | ✗ | 89.30% | 89.14% | 89.21% |
| LLaMA 3.1 8B | ✓ | 88.91% | 88.70% | 88.79% |
| Mistral 7B | ✗ | 89.40% | 89.34% | 89.35% |
| Mistral 7B | ✓ | 89.13% | 88.65% | 88.88% |

### *B. Common Issues of LLM-Generated Test Cases*

We manually analyzed the generated test cases and identified some common issues.

**Irrelevant Test Cases:** LLMs sometimes generate test cases that do not align with the specified use case making assumptions not supported by the scenario. This is especially apparent in the "Relevance" aspect in our human evaluation (Section III-D) and we can see how both GPT-4o and LLaMA suffer at this (Table VIII). This can be attributed to LLM's hallucination, a commonly known phenomenon where the model generates plausible-sounding but incorrect information. For example, for the use case "Organize Content into Columns", (where a user arranges content side-by-side by creating multiple columns on a note-taking application), GPT-4o generates a test case "Failed Content Organization Due to Invalid Column Number" to check a scenario where users might attempt an invalid column count. But such a scenario is not relevant in the given app's context.

**Redundant Test Cases:** LLM can produce test cases that are essentially duplicates (with slight modifications). They test similar scenarios that do not provide additional insight. For example, for the use case "Switch Between Delivery and Pickup While Browsing the Restaurant Menu" of an online food ordering app, LLaMA produces the following successful cases: i) Switch to Delivery, ii) Switch to Pickup, iii) Switch to Delivery from Pickup, iv) Switch to Pickup from Delivery. These test cases are intended to check successful transitions from the 'delivery' option to 'pickup' option and vice-versa. As such, the first and third test cases are the same and so are the second and fourth ones.

**Insufficient Test Cases:** LLMs may overlook test cases that human testers would typically consider, specially various edge cases. For example, for the use case "Add Icons and Cover Art" of a media management system, it is important to test that unsupported file-types are denied by the system. This edge case is incorporated in the human-generated test cases as "Invalid Cover Image type" where the test is performed with a 'pdf' file instead of an image (e.g., 'jpg', 'png'). However, both GPT-4o and LLaMA fail to include such edge cases.

**Lack of Input Specification:** LLM-generated test cases can sometimes lack a structured input format, which leads to inconsistent levels of detail and clarity compared to human-generated test cases. For example, in the use case "Add comments to Page" of a custom social media application, the human coder adds the following input fields for a given test case: ['user', 'commentText', 'mentionedPersons', 'mentionedGroups']. However, due to lack of proper context, the LLM-generated test cases fail to capture the 'mentionedGroups' field and only include the input fields: ['user', 'commentText', 'mention']. Moreover, human coders are often more capable utilizing realistic data entries for input fields compared to LLMS. While such realistic input values may seem unnecessary in the context of high-level test cases (specially when using dummy data), contextually appropriate entries can be helpful in certain cases. For example, in a mapping application, realistic place names and locations enable testers to understand the functionalities under test more accurately. They provide a meaningful reference for expected tests and results.

### *C. Threats to Validity*

***Internal validity*** concerns the potential inconsistencies within the study. One threat is response bias, where participants may provide answers to correspond with social desirability or perceived expectations. To address this, we assured participants of the anonymity and confidentiality of their responses. Another threat can be related to hyperparameter optimization. The models used in our study depend on hyperparameters that can significantly influence performance. Given the vast search space, exhaustive hyperparameter tuning was resource-intensive and beyond our primary objective. While the current configuration achieves satisfactory results, further optimization could yield better results.

***External validity*** relates to the generalizability of our findings to broader contexts. The participants of both the pilot and the main survey are selected using the principle of convenience sampling from the professional network of the authors. This could incur a selection bias. In order to avoid this, we included a diverse set of practitioners across different roles, companies, countries, and experience levels. Though we selected real-world applications from diverse sectors (e.g., finance, education, social media) and incorporated original student projects to capture a wide range of testing scenarios, our dataset may not fully represent all industry scenarios.

***Construct validity*** addresses the extent to which the study measures what it intends to measure. A primary construct validity threat arises from the possibility that our survey responses do not fully capture the practical challenges software teams face in aligning testing with business requirements. We

mitigated this by designing specific survey questions based on pilot study findings and mapping them directly to our research questions, ensuring alignment with our study objectives. The second concern regarding construct validity is subjectivity in prompt tuning. Effective prompts engineering of LLMs are not standardized in software engineering, and hence, prompt configuration is prone to variability. We mitigate this by experimenting with different settings of prompts and iteratively finding the best ones. This again reduces the variability due to prompts, though further refinements are possible.

***Conclusion validity*** concerns the validity of the conclusions that is made from the results. To ensure consistency and reduce individual evaluator bias in human evaluation, we used two evaluators for each sample. We also measured interrater reliability using Cohen's Kappa, which provided an objective metric for agreement among evaluators. Additionally, we provided clear guidelines on evaluation criteria to the evaluators that ensure the reliability of our findings.

## V. Related Work

Several approaches have been explored for automatic test case generation, including randomization, search-based algorithms, and deep learning.

Earlier randomization-based approaches, such as Randoop [36], generate test cases based on feedback-directed random sequences of method calls, and adapt based on previous execution results. Search-based methods like EvoSuite [15] employ evolutionary algorithms to iteratively create and optimize test cases for the highest code coverage. Recent works such as AthenaTest [47], A3Test [2] frame test case generation as a neural machine translation task (Code→Testcase) and generate test cases for a given (Java) method.

More recently, LLMs have been successfully employed for several software testing tasks, including test case generation [20], [49]. Schafer et al. designed TestPilot, an adaptive LLM-based test generation tool, to generate unit tests of JavaScript APIs [41]. Codex [8] was also utilized to generate code and test cases from natural language descriptions of the Program under Test (PUT) [7], [27]. Chen et al. proposed ChatUniTest, an LLM-based automated unit test generation framework that incorporates an adaptive focal context mechanism to encompass valuable context in prompts and rectify errors in generated unit tests [9]. Dakhel et al. proposed Mutation Test case generation with Augmented Prompt (MuTAP) that improves LLM test generation through iterative prompt mutation [12]. SymPrompt [40] structured test suite creation with sequenced, code-aware prompts to improve accuracy and coverage. TestART [18] utilized ChatGPT-3.5 based on prompt injection and feedback to improve the reliability of tests and high pass rate, high coverage, thus further demonstrating the potential of LLM-driven testing methods.

Even though LLMs have garnered significant engagement in generating test cases, high-level test case generation using LLMs is a field greatly under-explored, even though such test cases see great usage (Section II-C). Our study addresses this gap by thoroughly exploring the effectiveness of pre-trained LLMs and fine-tuned smaller LLMs in generating high-level test cases. Furthermore, utilizing only the use case of a task to generate test cases through LLMs is also a novel direction. Moreover, unlike most existing techniques (that require code to generate test cases), our proposed dataset and method (leveraging only use cases) supports early testing or test driven development, a key element of many agile frameworks [3].

## VI. Conclusion and Future Work

Our pilot survey of three local software companies points out to one of the most pivotal challenges in software testing: misinterpretation of business requirements and inadequate testing of vital functionalities. To address this, we assessed the potential of high-level test cases: test cases that primarily outline what functionalities and behaviors need to be tested, relaxing focus on the test-case-scripting part. We conducted another broader survey on a diverse collection of software industry experts to understand how helpful or necessary they find these test cases. The survey strongly validated our previous finding that a lack of understanding is the major challenge in software testing. We also discovered that majority of experts rely on high-level test cases and find them useful. They also showed interest in adopting an automated tool for high-level test cases, given that proper confidentiality was ensured.

We rigorously explored the effectiveness of large language models (such as GPT-4o and Gemini) for automatically generating high-level test cases. To address the requirement of confidentiality, we also evaluated fine-tuned smaller open-source models that can be maintained locally. For the evaluation and fine-tuning, we developed a novel dataset containing 3606 test that correspond to 1067 use cases of 168 diffent projects.

To further verify the quality of our curated dataset and the performance of the LLMs, another human survey was performed with the help of software industry experts. The participants evaluated the quality of the test cases generated by humans, pre-trained LLMs, and fine-tuned smaller LLMs on different criteria (e.g, Readability, Correctness, Completeness, Relevance, Usability). While the human-written test cases obtained the highest ratings, scoring very high (4+ out of 5) in each criterion, the LLM-generated ones also received satisfactory scores.

In the future, we plan to– i) Analyse LLM performance on challenging project types. We will identify areas where LLMs underperform, especially in complex or high-stakes applications, to guide model and prompt refinement, ii) Explore domain-specific fine-tuning We will fine-tune LLMs using project/domain-specific datasets, such as healthcare, or e-commerce, to improve the generation of relevant test cases for specific industries, iii) Focus on edge cases. Curriculum learning [31] can be explored in introducing simple cases first and progressing to more complex edge cases to improve the model's robustness, iv) Explore requirement generation from project summaries. We will leverage (and possibly enrich) our dataset to evaluate LLMs in generating detailed requirements and test cases from brief project descriptions, accelerating the initial stages of development.

## REFERENCES


[1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti. A3test: Assertion-augmented automated test case generation. *Information and Software Technology*, 176:107565, 2024.

[3] D. Astels. *Test driven development: A practical guide*. Prentice Hall Professional Technical Reference, 2003.

[4] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE transactions on software engineering*, (12):1278–1296, 1987.

[5] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*, pages 85–103. IEEE, 2007.

[6] T. B. Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[7] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[9] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 572–576, 2024.

[10] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[11] R. D. Craig and S. P. Jaskiel. *Systematic software testing*. Artech House, 2002.

[12] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, 171:107468, 2024.

[13] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.

[14] G. D. Everett and R. McLeod Jr. *Software testing: testing across the entire software development life cycle*. John Wiley & Sons, 2007.

[15] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[16] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023.

[17] V. Garousi, M. Felderer, M. Kuhrmann, K. Herkiloğlu, and S. Eldh. Exploring the industry's challenges in software testing: An empirical study. *Journal of Software: Evolution and Process*, 32(8):e2251, 2020.

[18] S. Gu, C. Fang, Q. Zhang, F. Tian, and Z. Chen. Testart: Improving llm-based unit test via co-evolution of automated generation and repair iteration. *arXiv preprint arXiv:2408.03095*, 2024.

[19] M. J. Harrold. Testing evolving software. *Journal of Systems and Software*, 47(2-3):173–181, 1999.

[20] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*.

[21] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. De Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly. Parameter-efficient transfer learning for nlp. In *International conference on machine learning*, pages 2790–2799. PMLR, 2019.

[22] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[23] A. Islam, N. T. Hewage, A. A. Bangash, and A. Hindle. Evolution of the practice of software testing in java projects. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 367–371. IEEE, 2023.

[24] M. A. Islam, M. E. Ali, and M. R. Parvez. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 4912–4944*.

[25] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[26] D. Kim, C. Park, S. Kim, W. Lee, W. Song, Y. Kim, H. Kim, Y. Kim, H. Lee, J. Kim, et al. Solar 10.7 b: Scaling large language models with simple yet effective depth up-scaling. *arXiv preprint arXiv:2312.15166*, 2023.

[27] S. K. Lahiri, S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, M. Musuvathi, P. Choudhury, C. von Veh, J. P. Inala, C. Wang, et al. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950*, 2022.

[28] B. Lester, R. Al-Rfou, and N. Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

[29] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[30] C.-Y. Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[31] Y. Liu, J. Liu, X. Shi, Q. Cheng, Y. Huang, and W. Lu. Let's learn step by step: Enhancing in-context learning ability with curriculum learning. *arXiv preprint arXiv:2402.10738*, 2024.

[32] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[33] Q. Luo, Y. Ye, S. Liang, Z. Zhang, Y. Qin, Y. Lu, Y. Wu, X. Cong, Y. Lin, Y. Zhang, et al. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *arXiv preprint arXiv:2402.16667*, 2024.

[34] M. L. McHugh. Interrater reliability: the kappa statistic. *Biochemia medica: Biochemia medica*, 22(3):276–282, 2012.

[35] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[36] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

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

[38] B. Potter and G. McGraw. Software security testing. *IEEE Security & Privacy*, 2(5):81–85, 2004.

[39] R. H. Rosero, O. S. Gómez, and G. Rodríguez. 15 years of software regression testing techniques—a survey. *International Journal of Software Engineering and Knowledge Engineering*, 26(05):675–689, 2016.

[40] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering*, 1(FSE):951–971, 2024.

[41] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 2023.

[42] N. Shinn, F. Cassano, A. Gopinath, K. R. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[43] H. Tang, K. Hu, J. P. Zhou, S. Zhong, W.-L. Zheng, X. Si, and K. Ellis. Code repair with llms gives an exploration-exploitation tradeoff. *arXiv preprint arXiv:2405.17503*, 2024.

[44] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[45] G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love, et al. Gemma: Open

models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.

[46] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[47] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*, 2020.

[48] S. Vanbelle. A new interpretation of the weighted kappa coefficients. *Psychometrika*, 81(2):399–410, 2016.

[49] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 2024.

[50] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint*, 2024.

[51] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi. Bertscore: Evaluating text generation with bert. In *8th International Conference on Learning Representations (ICLR)*, 2020.

[52] A. Zhou, K. Yan, M. Shlapentokh-Rothman, H. Wang, and Y.-X. Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.