Evolving Prompt with Genetic Algorithm Yields Powerful Instruction for Unit Test Generation

Pawit Wangsuekul* KAIST pawit_w@kaist.ac.kr
Hai-Nam V. Cao* KAIST hainam@kaist.ac.kr
Sorn Chottananurak* KAIST sorn111930@kaist.ac.kr
Thanh-Long V. Le* KAIST thanhlong0780@kaist.ac.kr

Abstract

Unit testing is a crucial aspect of software engineering, demanding considerable time and effort. To address this challenge, various automated test generation tools have been developed, such as ChatUniTest (Xie et al., 2023) — a ChatGPT-based system developed under the Generation-Validation-Repair framework. Despite its utility, ChatUniTest's performance is hampered by the reliance on manually crafted system prompts to initiate the test generation process. Drawing inspiration from recent researches in prompt evolution, we introduce EvolveUniTest, an adaptation of ChatUniTest with EvolvePrompt, a framework employing genetic algorithms for prompt evolution. EvolvePrompt initiates from a population of system prompts, including those manually designed for ChatUniTest, iteratively generating new prompts using a large language model and enhancing the population based on a development set. Leveraging the most optimized prompt from this evolutionary process, EvolveUniTest surpasses ChatUniTest in the quality and performance of generated unit tests. It achieves higher correctness percentages, increased branch and line coverage, and improved focal method coverage. Our code is available at: https://github.com/s6007541/EvolveUniTest.

1 Introduction

In today's complex software environment, even minor issues can lead to significant financial losses and damage the reputation of major companies. As a result, software testing, particularly unit testing, plays a vital role in ensuring high-quality software before launch. Yet, manual unit test creation is time-consuming, often causing developers to overlook this crucial task.

To address this challenge, automated tools for generating unit tests have emerged, falling into two categories: traditional tools and those using deep learning techniques. EvoSuite (Fraser and Arcuri, 2011), a traditional tool, employs evolutionary algorithms to create test cases, achieving high code coverage but generating tests that differ significantly from human-written ones. In contrast, AthenaTest (Tufano et al., 2021) and A3Test (Alagarsamy et al., 2023), deep learning-based tools, aim for more accurate testing through direct translation of source code, showing improved accuracy and method coverage compared to traditional tools.

There have been some recent advancements to leverage large language models (LLMs) for unit test generation. By incorporating LLMs into this context, developers can benefit from improved efficiency, accuracy, and scalability in generating unit tests for their software applications. ChatTester (Yuan et al., 2023) and ChatUniTest (Xie et al., 2023) are the unit test generation approaches based on LLM which utilizes ChatGPT to enhance the quality of the generated tests. While surpassing EvoSuite in coverage and outperforming AthenaTest and A3Test in method coverage, these LLM-based system rely on manually crafted prompts, potentially limiting its optimal generation performance.

Evolutionary algorithms are algorithms inspired by the principles of natural evolution. (Liu et al., 2023) reveals the great potentials of LLMs as evolutionary optimizers for solving combinatorial problems with LLM-driven EA. Prior research has attempted to integrate evolutionary algorithm for prompt evolution. An example of such an approach is Promptbreeder (Fernando et al., 2023), a self-referential self-improvement method designed for LLMs. This technique evolves prompts tailored to a specific domain while simultaneously enhancing the prompt evolution process itself.

To address the issues mentioned of manually crafted prompts, we propose our pipeline, **EvolveUniTest** integrated with EvolvePrompt, an evolu-

*Equal contribution.

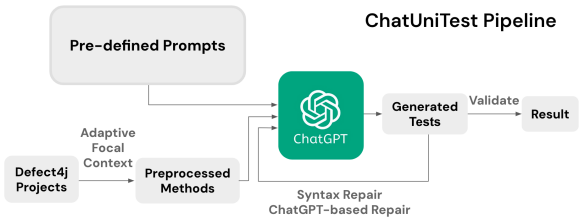

Figure 1: ChatUniTest pipeline showing Generation-Validation-Repair framework

tionary prompt optimization for generating unit tests. Our approach is inspired by EvoPrompt (Guo et al., 2023), an evolutionary algorithm for prompt optimization. Our system iteratively creates fresh prompts using a language model and enhances the prompt collection using a development dataset. By harnessing the most refined prompt from this evolutionary method, EvolveUniTest surpasses ChatUniTest in the quality and performance of produced unit tests. Empirical evidence demonstrates that EvolveUniTest achieves superior correctness rates, expanded branch and line coverage, and enhanced focal method coverage.

2 Related Work

ChatUniTest (Xie et al., 2023) is an automated unit test generation tool based on ChatGPT developed within the Generation-Validation-Repair framework as shwon in Figure 1. Existing tools like EvoSuite (Fraser and Arcuri, 2011) and AthenaTest (Tufano et al., 2021) have limitations, either in readability or correctness of generated unit tests. ChatUniTest parses projects, creates a context around focal methods and dependencies within token limits, then submits this context to ChatGPT to generate tests. It extracts and validates responses, using rule-based repair for simple errors and ChatGPT-based repair for complex issues. Evaluation shows that ChatUniTest outperforms EvoSuite in branch and line coverage, surpasses AthenaTest (Tufano et al., 2021) and A3Test (Alagarsamy et al., 2023) in focal method coverage, and effectively generates assertions while handling complexities like mock objects and reflection for test objectives.

EvoPrompt (Guo et al., 2023) is a framework aimed at automating the creation of prompts for Large Language Models (LLMs) by leveraging evolutionary algorithms (EAs). Instead of relying on manually crafted prompts, EvoPrompt uses EAs to optimize natural language prompts for coherence and readability while connecting with LLMs. This synergy allows for prompt optimization without gradients or parameters. The approach iteratively generates and improves prompts based on LLM outputs, surpassing human-engineered prompts and existing methods in performance across various language understanding and generation tasks. Additionally, EvoPrompt showcases the potential synergy between LLMs and conventional algorithms, potentially inspiring further research in this domain.

3 Method

In this section, we first briefly go through an overview of EvolveUniTest. Then, we discuss our main contribution in adapting ChatUniTest into EvolveUniTest with two main stages: the integration of open-sourced LLM and the implementation of prompt evolution through genetic algorithm.

3.1 Overview of EvolveUniTest

The architecture of EvolveUniTest system largely follows the framework of the original ChatUniTest, with the addition of evolution stage right before the test generation process, which makes it a 4-stage Evolution-Generation-Validation-Repair framework as shown in Figure 2.

- **Evolution**: During the evolution phase, EvolveUniTest employ genetic algorithm to evolve an initial set of system prompts to one final optimized system prompt that would be used in the generation phase. To evaluate the quality of each system prompt in each generation's population, we manually designed a validation set of 16 methods, on which we calculate the metrics of unit tests generated from assessing system prompt. More details could be found in the below sub-section.
- **Generation**: In the generation stage, EvolveUniTest creates an adaptive focal context by considering the focal method and adhering to a predetermined maximum limit of prompt tokens. Subsequently, it transforms this context into a user prompt to query the LLM and retrieve the response.
- **Validation**: During the validation phase, EvolveUniTest extract the unit test (if there's any) from LLM's response and check if the test is syntactically sound, then try to compile and execute it. In case of any failure due to

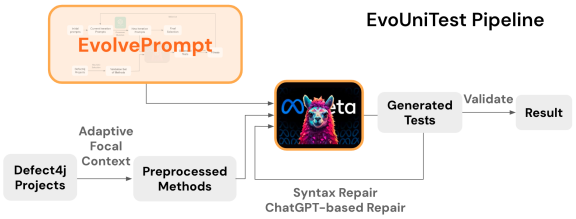

Figure 2: Our proposed EvolveUniTest pipeline developed on top of original ChatUniTest pipeline

syntax error, compiled error or runtime error, the test would be sent to the final stage of the frame work.'

- **Repair**: Any failed test case would be processed by the repair component of EvolveUniTest, starting with rule-based fixing, then proceed to LLM-based fixing in case the former failed.

3.2 Integration of Open-sourced LLM

The most important core component of EvolveUniTest is the underlying LLM, with the significant role of generating and repairing the test cases. In the original ChatUniTest system, the author used OpenAI API gpt-3.5-turbo to handle both of those tasks. This approach does not align well with our EvolveUniTest, since the implementation of GA inside EvolveUniTest means that the number of API calls would increase significantly to approximately 200 times bigger than that of ChatUniTest. As a result, we need to abandon the API call and with alternative implementation of open-sourced LLM. We shortlisted 3 families of LLM as our candidates: Llama, CodeLlama, and Falcon. Within our budget of computational resource, we decided to go with CodeLlama-7b-Instruct, which is smaller compared to other models, but still provides impressive capability in programming tasks and strong ability to follow human's instruction (the model was instructed-fintuned). Particularly, we develop a simple Flask server to host CodeLlama-7b-Instruct model and provide an API endpoint for querying LLM.

3.3 Implementation of Genetic Algorithm

We largely base our genetic algorithm on the EvoPrompt framework (Guo et al., 2023). However, one technical difficulty arises from the major distinction between pure natural language processing tasks versus software testing: the complexity of

Algorithm 1 Prompt Evolution through Genetic Algorithm

Require: Size of population N, initial system prompts $P_0 = p_1, ..., p_N$, methods dev set D, number of iterations T, genetic operator Ga, $f_D(.)$ denotes the fitness (based on generated unit tests) of system prompt.
$S_0 \leftarrow \{s_i = f_D(p_i) | i \in [1, N]\}$
for i from 1 to T **do**
  $P_{current} \leftarrow \emptyset$
  while $|P_{current}| < N$ **do**
    Sample p_{parent_1} and p_{parent_2} from P_{i-1} based on S_{i-1}
    $p_{child} \leftarrow Ga(p_{parent_1}, p_{parent_2})$
    $s_{child} \leftarrow f_D(p_{child})$
    $P_{current} \leftarrow \{P_{current}, p_{child}\}$
    $S_{current} \leftarrow \{S_{current}, s_{child}\}$
  end while
  $P_i \leftarrow \{P_{i-1}, P_{current}\}$
  $S_i \leftarrow \{S_{i-1}, S_{current}\}$
  $P_i \leftarrow$ Top 10 prompts in P_i based on S_i
  $S_i \leftarrow$ Top 10 scores in S_i
end for
$p* \leftarrow$ best prompt in P_T based on S_T
return $p*$

fitness function. Each pure NLP task has a predefined metric to evaluate the performance, such as perplexity for language modeling or F1 score and exact match score for question answering. In contrast, the task of software testing in our context does not have an inherent concrete metric, so we need to design our intuitive fitness function in order to evaluate the quality of each individual system prompt in the population with respect to the test generation quality. To this end, our fitness function is as follows. Denote *CTR* (correct test rate) as a percentage of successfully executed test files, *CMR* (correct method rate) as a percentage of successfully executed methods. *BC* and *LC* are average branch coverage and line coverage across all files respectively.

$$f(p) = \beta * (CTR + CMR) + \alpha * (BC + LC)$$

We additionally present two weight hyperparameters, where α is assigned a value of $\frac{1}{4}$ and β is set to 1. These values are chosen intuitively to indicate the relative significance of line coverage and branch coverage.
The overview of our proposed genetic algorithm is

shown in Figure 3. Below is a detailed description of major aspects in the prompt evolution algorithm:

- **Initial Population**: We take the hand-crafted system prompt from ChatUniTest and pass it through ChatGPT to generate N different variations based on paraphrasing, resulting in the initial population of system prompts P_0. We ensure that these N variations share similar content while featuring distinct sentence structures and wording.
- **Fitness computation**: We evaluate each individual system prompt based on our designed metric $f_D(p)$. This essentially means that we have to run the whole unit test generation pipeline on the development set to compute the fitness of each individual, which could be quite time-consuming and resource-demanding. To address this challenge, we only ran one round of the repair phase in the original generation pipeline for syntactical errors.
- **Selection**: To create new generation of system prompts, we choose two parents from the current population based on the roulette wheel selection method according to their fitness on the development set. If a prompt p_i generate a set of unit tests that achieve a score of s_i on the development set, the probability of that prompt being chosen as one of the parents is $s_i / \sum_{j=1}^{N} cost(s_j)$.
- **Evolution**: This step includes crossover and mutation, both of which are conducted through LLM. In crossover, we ask LLM to extract and combine different components from the parent prompts to create a new individual prompt. After that, this candidate prompt goes through the mutation process, with random modifications being made to its content. We utilize the gpt3.5-turbo model from OpenAI API to perform this operation.
- **Generational Selection**: The system iteratively generates new prompts and compute their fitness on the development set, until the current set of candidates reach $2N$. Finally, only the top N best prompts in the current set are retained and become the official new generation of system prompts.

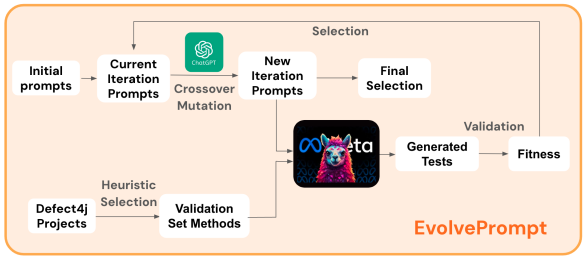

Figure 3: Our evolution algorithm draws inspiration from EvoPrompt in the pursuit of creating an optimal prompt for the unit-test generation task.

4 Experiment

4.1 Dataset

We will employ our pipeline to automatically generate unit tests for four Java projects listed in Table 1: Gson, Lang, Csv and Cli. These projects were previously chosen in the ChatUnitTest pipeline (Xie et al., 2023), and we utilize them for performance comparison.

4.2 Baseline

We will establish the ChatUnitTest's performance on all four datasets as our baseline. From this baseline, we identify certain limitations in the prompting technique of previous work, as evidenced by numerous tests requiring additional rounds of correction. This observation serves as motivation for us to enhance the prompt quality with the evolution algorithm.

4.3 Design

In this experiment, we utilized an open-source LLM known as Code Llama-7b-Instruct [1]. The experiments were conducted using an NVIDIA GeForce RTX 3090. We organized the experiment into two phases. In Phase 1, we applied an evolution algorithm to a specific set of files from the Gson project. More specifically, we tested 16 candidate methods from Gson project. The best prompt identified in Phase 1 was used to validate the performance on the unit test generation pipeline across additional datasets such as Lang, Csv, Cli.

4.4 Hyper-parameters

EvolveUniTest Pipeline: Here are the hyper-parameters utilized within the EvolveUniTest pipeline. We employ a single process concurrently due to our bottleneck, which lies within the LLM

[1] facebookresearch/codellama

Projects	System	Methods	Attempts	Invalid	CompileError	RuntimeError	Correct
Cli	ChatUnitTest	250	750	4	479	189	78
	Ours	250	750	2	462	195	**91**
Csv	ChatUnitTest	213	639	316	236	45	**42**
	Ours	213	639	325	216	57	41
Gson	ChatUnitTest	660	1980	375	1133	268	204
	Ours	660	1980	369	1121	270	**220**
Lang (NumberUtils)	ChatUnitTest	67	201	44	22	39	99
	Ours	67	201	21	24	27	**129**

Table 1: Quality of Generated Test Cases.

Project	Correct Test Cases (%)				Focal Method Coverage (%)			
	A3Test	AthenaTest	ChatUnitTest	Ours	A3Test	AthenaTest	ChatUnitTest	Ours
Cli	25.2	11.1	10.4	**12.1**	37.2	29.5	23.0	**25.2**
Csv	25.7	9.0	**6.6**	6.5	37.8	34.3	12.2	**12.9**
Gson	14.1	2.9	11.0	**11.1**	40.9	9.5	20.6	**23.1**
Lang (NumberUtils)	-	-	49.3	**64.2**	-	-	74.6	**83.6**

Table 2: Quality of Generated Test Cases compared to State-Of-The-Art

call. Utilizing multiple concurrent processes for unit test generation wouldn't enhance our pipeline's runtime performance. For each method, we generate three unit tests and set the maximum round of the repair process to double the default value. The maximum prompt tokens are configured at 2,048, while the minimum error tokens adhere to the specifications outlined in the original ChatUniTest paper.

Genetic Algorithm: For hyper-parameters related to our GA experiments, we chose a population size of 10, 5 generations, and 16 methods to run on because of limited time and resources. Evaluating each individual system prompt essentially means that we have to run the whole unit test generation pipeline on the development set to compute the fitness of each individual. With this setting, the whole process of evoprompt phase alone would take us at least 7 to 8 hours to run.

Code Llama-7b-Instruct: Regarding our Code Llama-7b-Instruct, due to limited computational power, our LLM hyper-parameters aren't maximized. The maximum sequential length is set at 2,048 due to GPU memory constraints. Expanding this length with additional GPU memory would notably reduce the number of generated test cases that become invalid due to aborted processes or syntax errors. We've set the maximum batch length at 4, with a temperature of 0.2, while the top p value is established at 0.95. Frequency penalty and presence penalty are both set to 0.

ChatGPT: In terms of the ChatGPT's hyper-parameters used during prompt evolution, we utilized the gpt-3.5-turbo model with a temperature of 0.5 and top P value of 0.95. Both frequency penalty and presence penalty were configured at 0.

4.5 Evaluation Metrices

In evaluating the quality of the generated test cases, we considered several factors, including:

- **Syntactic Correctness** The concept of syntactic correctness involves ensuring that the test code aligns with Java syntax rules, a verification process that will be conducted through a Java parser.
- **Compile Correctness** Compile correctness will be verified if the test generates no errors during compilation through the Java Compiler.
- **Passing** We'll employ JUnit as the test executor to ascertain a test's success, ensuring it doesn't yield any runtime errors during execution.
- **Testing APIs invocations** Effectively crafted unit test cases ought to confirm the anticipated behavior of the MUT (Method Under Test) by utilizing Testing APIs. For example, these

cases should leverage JUnit Assert APIs (such as assertEqual(), assertTrue()) to validate the return value of the MUT and employ Mockito Framework APIs (like mock(), verify(), when()) to emulate the behavior of an external dependency.

- **Correct** A test is considered valid when it demonstrates syntactic accuracy, compiles and executes without errors, calls the method under test, includes assertions, and facilitates the successful passage of the test by the target code.

We assess the comparison between ChatUniTest and our approach across four Java projects, considering branch and line coverage. In evaluating AthenaTest, A3Test, ChatUniTest, and our method, we'll analyze their respective percentages of correct test cases and focal method coverage across the same 4 projects from Defects4J dataset.

- **Correct Test Percentage** The correct test percentage indicates the proportion of accurately formulated tests among all attempts made.

- **Focal Method Coverage** Focal method coverage represents the percentage of focal methods covered while generating unit tests. A focal method is deemed covered if it was accurately tested within the specific attempt.

5 Results & Discussion

Compare with ChatUniTest In Table 1, we present EvolveUniTest's performance against our baseline, ChatUniTest, across 4 Java projects. This involved processing a total of 1,190 methods and conducting 3,570 unit test generation attempts. Among these attempts, ChatUnitTest encountered 739 invalid unit tests, where they were aborted due to exceeding the maximum prompt token limit or encountering syntax errors. On the contrary, EvolveUniTest managed to reduce the count of invalid unit tests to 717. Across the projects, Cli and Lang exhibited a lower number of invalid test cases compared to others. Notably, within these two projects, our proposed EvolveUniTest led to the most substantial improvement in the number of final correct unit test cases compared to the original ChatUniTest. Manual investigation revealed that the primary reason behind these errors was the generation of excessively lengthy tests, which were subsequently truncated and couldn't be repaired by the syntactic repair component. This issue could potentially be resolved by providing a larger GPU memory space and expanding the token limit lengths of the LLMs used in the pipeline. Therefore, we anticipate that with increased computational power, EvolveUniTest could significantly reduce the number of invalid test cases and bring about substantial enhancements, particularly in the Csv and Gson projects. Compile errors were a prevailing occurrence in the majority of unit test generation attempts made by both ChatUniTest and EvolveUniTest. The most frequent error types included "cannot find symbol" or "incompatible types," mainly attributed to ChatUniTest generating non-existing methods or classes or making errors in their utilization. Regarding runtime errors, the prevalent types stemmed primarily from misusing assertions or reflection. In terms of passed and correct test cases, our proposed EvolveUniTest showcased significant superiority over ChatUniTest in all projects except for the Csv project, where EvolveUniTest yielded a comparable number to ChatUniTest. This highlights the potential of EvolveUniTest across various projects, showcasing commendable performance irrespective of project size or Java version when compared to the baseline. This overall performance suggests that EvolveUniTest could serve as an alternative method to ChatUniTest for generating test cases across a broad spectrum of Java projects.

Branch and Line Coverage In Table 3, We've conducted a comparison between the branch and line coverage achieved by ChatUniTest and our proposed EvolveUniTest across four distinct Java projects. Upon reviewing the table, it's evident that EvolveUniTest generally outperforms ChatUniTest concerning both line and branch coverage across the majority of projects. For instance, in the Cli project, EvolveUniTest attains higher line and branch coverage percentages (54.1% and 44.6%, respectively) compared to ChatUniTest (52.6% and 38.3%, respectively). This trend remains consistent in other projects like Csv, Gson, and Lang. However, there are instances where ChatUniTest surpasses EvolveUniTest in line coverage but with slight advantages. For example, in the Gson project, ChatUniTest achieves a higher branch coverage of 29.9% compared to EvolveUniTest's 28.2%. Despite these occasional exceptions, EvolveUniTest consistently outperforms ChatUniTest in terms of line coverage across all projects. On average,

EvolveUniTest demonstrates superiority in both branch and line coverage, with an average line coverage of 48.6% compared to ChatUniTest's 43.3%, and an average branch coverage of 33.2% compared to ChatUniTest's 28.5%. This consistent performance showcased by EvolveUniTest underlines its reliability and effectiveness as an alternative tool for unit test generation.

Project	Line Coverage (%)		Branch Coverage (%)	
	ChatUnitTest	Ours	ChatUnitTest	Ours
Cli	52.6	**54.1**	38.3	**44.6**
Csv	26.4	**40.0**	6.7	**19.7**
Gson	36.8	**38.9**	**29.9**	28.2
Lang (NumberUtils)	57.4	**61.2**	38.9	**40.2**

Table 3: Performance of Generated Test Cases

Compare with other SOTAs In Table 2, we present an assessment of the performance of four tools across the Defects4J dataset's four projects, focusing on the percentage of correct test cases and focal method coverage. Our comparison encompasses the evaluation of these tools within the Defects4J dataset. The data pertaining to AthenaTest and A3Test in the table is derived from A3Test (Alagarsamy et al., 2023). Upon reviewing the table, it's evident that our proposed EvolveUniTest consistently outperforms ChatUniTest across most of the dataset concerning the percentage of correct test cases and focal method coverage. Despite utilizing a less potent LLM, our EvolveUniTest also demonstrates relatively similar performance to AthenaTest. However, A3Test significantly outperforms our EvolveUniTest in all projects in both percentage of correct test cases and focal method coverage. Nevertheless, as per the original ChatUniTest paper, given an equivalently powerful LLM with GPT3.5, we anticipate outperforming both A3Test and AthenaTest in all projects concerning both metrics.

6 Conclusion

In this project, we introduced EvolveUniTest, an innovative unit test generation system utilizing LLMs and incorporating the features of the existing ChatUniTest alongside our genetic algorithm, EvolvePrompt. Through extensive empirical investigations involving diverse real-world projects, our system demonstrated promising outcomes, surpassing other contemporary systems in terms of both generated test quality and performance. This underscores the potential application of genetic algorithms, particularly in the context of evolutionary algorithms, within the realm of unit test generation. With enhanced resources and further refinement of our evolutionary algorithm, we anticipate overcoming current limitations and achieving or surpassing the benchmarks set by state-of-the-art systems in unit test generation.

References

Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2023. A3test: Assertion-augmented automated test case generation.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2023. Promptbreeder: Self-referential self-improvement via prompt evolution.

Gordon Fraser and Andrea Arcuri. 2011. Evosuite: Automatic test suite generation for object-oriented software. pages 416–419.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. 2023. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers.

Shengcai Liu, Caishun Chen, Xinghua Qu, Ke Tang, and Yew-Soon Ong. 2023. Large language models as evolutionary optimizers.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit test case generation with transformers and focal context.

Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. Chatunitest: a chatgpt-based automated unit test generation tool.

Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation.