Influence of External Dependency Retrieval and Prompt Engineering in Test Case Generation using LLMs

David Lenke, Javier Ferrer*, and Francisco Chicano

ITIS Software, Universidad de Málaga, Spain
lenke@uma.es, jferrer@uma.es, and chicano@uma.es
https://neo.lcc.uma.es

Abstract. The recent rise of large language models (LLMs) has enabled the generation of higher-quality test cases by leveraging the semantics of the methods under test. However, existing LLM-based approaches still struggle to achieve high coverage levels. To mitigate this issue, we present two complementary techniques in this work: Prompt Engineering and External Dependency Retrieval for context enrichment. We evaluated our improvements through an ablation study on three open-source and four proprietary projects, encompassing 261 distinct methods. For each method, we generated test suites under four implementations and performed ten independent runs, yielding a total of 10,440 executions. Our combined approach yields an average coverage increase of 12% on industrial software, with statistically significant gains over all other variants studied in this paper. Although our enhancements increase the context (the number of input tokens rises by 66.3%), this is partially compensated by a reduction in output tokens due to fewer repair attempts, so that the overall cost overhead remains moderate at about 16%. As future work, we aim to identify the minimal necessary context that still yields significant improvements in test coverage, which could help to further reduce costs.

Keywords: Test Case Generation · LLM · External Dependency Retrieval· Prompt Engineering· ChatUniTest

1 Introduction

Automatic test case generation remains a great challenge for software engineers. Despite improvements in techniques like search-based software testing [5], test cases generated are not comparable to those written by humans. The generated tests often lack readability and do not include meaningful assertions, which made them not very useful. However, the recent adoption of large language models (LLMs) in software engineering tasks has led to significant improvements in both aspects [1, 9]. But most LLM-based techniques for the automatic generation of

* Corresponding author

test cases are not achieving the highest coverage compared to state-of-the-art techniques [8], thus there is still room for improvement.

A wide range of test case generation tools have been proposed in recent years. One of the most prominent is EvoSuite [5], a tool based on genetic algorithms that has traditionally been used for automated test case generation. However, it has some drawbacks due to its lack of semantic understanding of the code. As a result, the tests generated are sometimes hard to read and maintain. Another widely used tool is Randoop [4], which employs randomness to generate test cases but tends to generate repetitive and irrelevant tests due to the absence of a specific coverage objective. AthenaTest [6] is another tool which uses its own trained transformers model to generate the tests. Nevertheless, it often struggles generating correct test cases because of its training limitations.

More recently, the advent of LLMs has enabled a new generation of test generation tools, offering an alternative to traditional approaches. Among these new tools, we can find ChatTester [9], which improved ChatGPT for test case generation; ChatUniTest [1], which provides a framework to add different test case generator implementations; and HITS [8], a tool that generates test cases using program slices focused on the method under test. However, these tools present some limitations, as they rely completely on the LLM's internal knowledge of the external dependencies, which the language model may not be trained on. This can prevent the generation of meaningful object inputs needed to exercise different code paths and increase coverage. To address these issues, we propose incorporating two complementary techniques to a new implementation based on ChatUniTest.

The context provided to the LLM is key for generating more meaningful tests, since it gives the model insight into the internal logic and can thus help achieve higher coverage. When external or unknown dependencies appear in the code, results can be poor because the LLM tends to produce very generic test cases. For this reason, we believe that including the code of external dependencies could improve both coverage and quality of the generated tests. In this work, we enrich the prompt with additional code from external dependencies to enhance automatic test case generation using LLMs.

Beyond the extra context the LLM needs, it also requires clear instructions. We have found that optimizing the prompt can improve the results. For this reason, another improvement we plan to evaluate in this paper is the potential coverage gains from an optimized prompt. We will explore both approaches using ChatUniTest as our base. We selected ChatUniTest because it is a novel framework designed to simplify extensions around the core LLM-based test generator.

The challenge we address in this paper is particularly relevant to industrial software. On the one hand, such systems need high test coverage to support reliable development. On the other, they often depend on a large number of private third-party libraries. Since this private code has not been included in the training data of existing LLMs, models struggle when those dependencies are missing. That is why we evaluate our approach on both open-source projects previously used to compare well-known tools such as EvoSuite [5], ChatTester [9],

ChatUniTest [1], and HITS [8], as well as entirely private code provided by an industrial partner.

We formulate the following research questions to guide our evaluation:

- RQ1: Do the proposed improvements lead to higher branch coverage?
- RQ2: Does increasing the amount of context result in higher test generation cost?

The rest of this article is organized as follows. Section 2 presents the proposed methods and the baseline used for comparison. In Section 3, we describe the experimental setup, algorithms, and dataset used in our experiments. Section 4 discusses the results and addresses the research questions. Finally, Section 5 presents our conclusions and outlines future work.

2 Method

2.1 Baseline

ChatUniTest [1] is an automatic test case generation tool that relies on LLMs. It achieves its objective by using different techniques, such as *Adaptive Focal Context* or the *Generation-Validation-Repair* mechanism, both of which are proposed in this work as innovative approaches. *Adaptive Focal Context* provides the prompt with information about the focal method while excluding worthless information about the focal class. Meanwhile, the *Generation-Validation-Repair* mechanism follows several phases: it first generates the test suite, then validates it, and, in case of compilation or execution errors, it performs a predefined number of repair rounds until all errors are fixed.

ChatUniTest is also a framework that allows the integration of test generation implementations. In this way, it serves as a hub of software tools for test case generation with LLMs. ChatUniTest consists of a core workflow called ChatUniTest Core. It also includes ChatUniTest Toolchain, which includes a Maven plugin that simplifies integration into software projects. ChatUniTest infrastructure can be seen in Figure 1. Our implementation was also added to this framework, making it easy to apply the enhancements proposed in this paper to other implementations based on ChatUniTest. Note that our source code is available as a fork of the original ChatUniTest Core[1] and ChatUniTest Maven Plugin[2] repositories.

2.2 External Dependency Retrieval

In our External Dependency Retrieval implementation, we automatically retrieve and decompile the source code of every Maven dependency used by the target method. By injecting this real dependency code into the prompt, we remove

[1] https://github.com/NEO-Research-Group/chatunitest-core.git
[2] https://github.com/NEO-Research-Group/chatunitest-maven-plugin.git

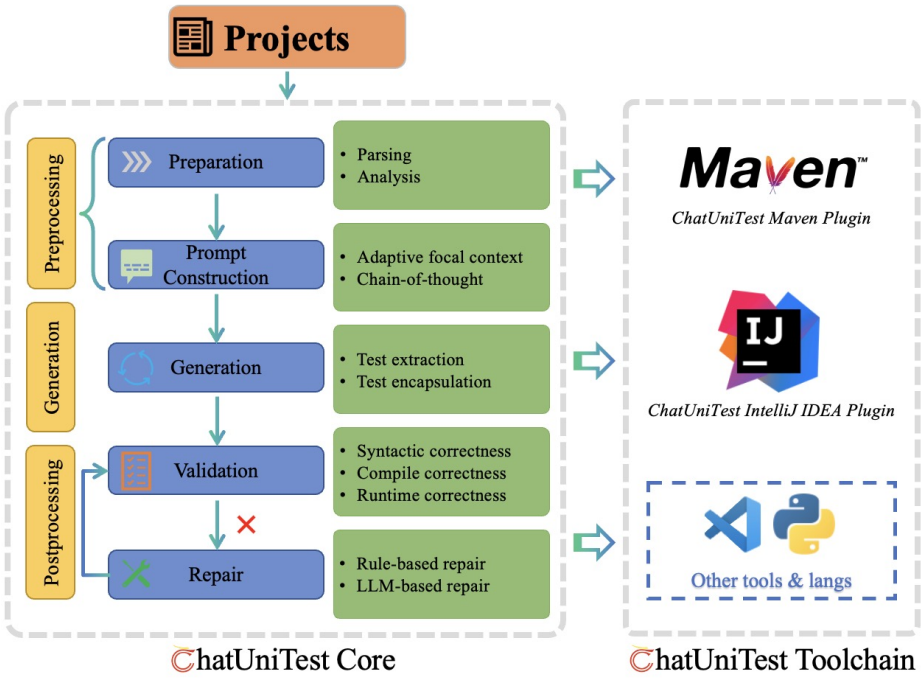

Fig. 1. This image shows the ChatUniTest workflow [1], having a Core module where all the logic is done and the plugins where it can be used.

the assumption that the LLM already knows library internals. This technique is innovative because other LLM-based test generators do not account for external dependencies in their prompts. The complete mechanism is shown in Figure 2 and described in detail in the following paragraphs.

To support effective test case generation, we extended ChatUniTest Maven plugin to extract and supply necessary dependency code to ChatUniTest Core. The customized plugin first scans the local Maven repository to identify the precise binaries for the target project's dependencies. It then decompiles each dependency and using the same criterion as for regular dependent classes, we add the complete source code of external classes to the prompt context.

Note that, to bridge the gap between binary representation and the source-level understanding required by the LLMs, our implementation incorporates a decompilation step. Specifically, each .class file needed is decompiled using CFR decompiler to recover an approximate version of the original Java source code. Although the decompiled code may not exactly match the original source, it typically retains the essential logic and structure needed to provide enough context to the LLM.

The recovered source code of the class is then integrated into the prompt template alongside the regular context provided by ChatUniTest. This prompt will

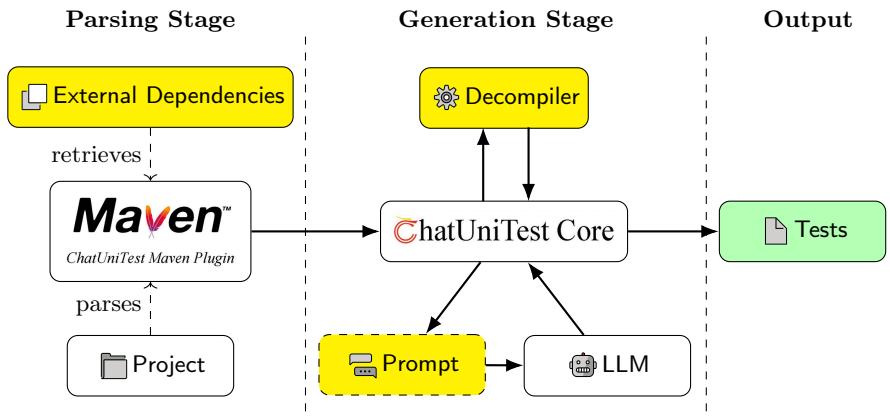

Fig. 2. External Dependency Retrieval in ChatUniTest environment. Characteristics involved in this work are in yellow, other components in white, output in green.

now provide meaningful information for the LLM to use the fields and methods of the classes implemented outside the target project. As a result, ChatUniTest no longer relies on the model's implicit knowledge of dependencies, instead it clearly shows how external methods are invoked and which fields they use.

2.3 Prompt Engineering

In this section, we describe our prompt engineering strategy aimed at generating higher-quality test cases. After reviewing the default ChatUniTest prompts, we identified areas for improvement and applied targeted techniques to produce prompts that guide the LLM toward more precise test generation. In the following, we describe two techniques to improve the prompt.

Few-shot prompting [2]: The Few-shot prompting technique consists of adding multiple examples to a prompt to guide the LLM to perform a specific task. In our case, we added two examples: one showing how the LLM should handle accessing and modifying private fields, and another illustrating the invocation of private methods.

Prompt Self-refine [3]: Prompt Self-Refine is a technique in which the LLM is used to iteratively improve its own prompt. In the default ChatUniTest setup, the system prompt is concise but lacks explicit guidance for generating test cases that achieve high coverage. To address this, we applied Prompt Self-Refine resulting in an enhanced version of the prompt that includes strategic instructions aimed at maximizing test coverage.

First, we refactored the original system prompt for test case generation into a Markdown version and added some constraints. Then, the refinement was carried out in two steps. In the first step, we asked the LLM to 'Improve it using prompt engineering standards'. In the second step, we further refine the result by giving to the model the following prompt: 'Improve the part of getting high coverage. I

want you to create new requirements if needed to get more coverage', using the output from the first step as input. The final prompt templates are available in our GitHub repository.

3 Experiments

3.1 Experimental Setup

All experiments were conducted on a workstation equipped with Intel Core i7-7700K CPU, 16 GB RAM, Intel Corporation HD Graphics 630 GPU, running Ubuntu 22.04 with Linux kernel 5.15. The software environment included Java 11.0.25, Maven 3.9.9 and ChatUniTest Core and ChatUniTest Maven plugin available at 26 of February of 2025. We used Codestral-2501[3] large language model to generate the test cases.

Despite using recommended LLM parameters such as a zero temperature and a fixed random seed, all other settings remain at their default values. Nonetheless, the techniques presented in this paper remain stochastic, as identical prompts can yield different outputs across executions. To ensure a fair and robust comparison, we perform 10 independent runs for each configuration. It is worth noting that this variability is often overlooked in existing LLM-based approaches for test case generation.

We analyzed the performance of every configuration through multiple ablation studies comparing the branch coverage results obtained across 10 independent iterations using identical parameters. Statistical significance was assessed by computing the p-values in the non-parametric Wilcoxon signed-rank and the Wilcoxon rank-sum tests for each pairwise comparison, a total of $m = 6$. To maintain an overall significance level of $\alpha = 0.05$, we applied Bonferroni correction to account for multiple hypothesis testing, resulting in an adjusted significance threshold of $\alpha_{\text{corrected}} = 0.0083$. This approach ensures a confidence level of 95% while reducing the risk of Type I errors (True or False positive).

In addition, with the aim of properly interpreting the results of statistical tests, it is always advisable to report effect size measures. For that purpose, we have also used the non-parametric effect size measure $A12$ statistic proposed by Vargha and Delaney [7]. This statistic quantifies the probability that one algorithm outperforms another, providing insight into the practical relevance of the observed differences.

3.2 Implementation

In this work, we aim to measure the benefits of the proposed techniques. To this end, we conducted an ablation study that allows us to evaluate the impact of each technique both in isolation and in combination. The compared implementations are: ChatUniTest default implementation (CUT), ChatUniTest with Prompt Engineering (PO), ChatUniTest with External Dependency Retrieval

[3] https://mistral.ai/news/codestral-2501

(*EDR*) and both External Dependency Retrieval and Prompt Engineering together (*ALL*).

The same parameter values were used for all the four implementations. Regarding the parameters, we used a set-up where only compilation errors were repaired with a maximum of five trials. The number of test suites generated per method was fixed to one. After the generation process, the resultant tests were executed and the coverage value was obtained using Jacoco[4], a tool for getting coverage information of tests.

3.3 Dataset

We selected some public and private Maven projects to perform the experiments. For the public projects, we selected three open-source GitHub projects that have been used in other studies with the same objective [8]. Specifically, the chosen projects were batch-processing-gateway[5] (bpg), JDom2[6] and windward[7]. In the case of the private projects, we performed the experiments in four Maven projects provided by an industrial partner. For a fair comparison, we selected only those methods that required at least one external dependency class in the context provided to the LLM. In addition, we restricted the evaluation to methods containing more than one branch, since our comparison is based on branch coverage. The benchmark summary can be seen at Table 1.

Table 1. Dataset statistics. Columns 5 to 9 show the methods unter test (MUTs), the average lines of code, number of branches, McCabe complexity, and external dependencies reliance for all the project methods, respectively.

Project	Domain	Version	Public	MUTs	Lines	Branches	Compl.	Ext. Dep.
JDom2	Text Processing	2.0.6.1	Yes	20	14.5	8.9	5.6	2.2
windward	Microservices	1.5.3	Yes	11	9.6	3.8	2.9	2.5
bpg	Cloud Computing	1.1	Yes	49	19.7	9.1	5.6	3.1
Project1	eGovernment	3.0.0	No	49	20.3	8.0	5.0	2.5
Project2	Library	3.0.0	No	1	5.0	2.0	2.0	1.0
Project3	Rule Engine	0.0.1	No	58	11.1	6.5	4.3	2.9
Project4	API Gateway	3.0.0	No	73	16.8	7.5	4.7	3.7

4 Results

To offer a detailed assessment of our proposals, we conducted an ablation study that evaluates each technique in isolation. We then combined the two techniques

[4] https://github.com/jacoco/jacoco
[5] https://github.com/apple/batch-processing-gateway
[6] https://github.com/hunterhacker/jdom/tree/master
[7] https://github.com/Flmelody/windward

to create and evaluate four different implementations. We then answer the research questions posed earlier by analyzing the experimental results.

4.1 Test Case Coverage

Table 2 presents the average branch coverage percentage obtained by every implementation for all the methods in public and private projects and its standard deviation. All implementations improved the average branch coverage over the baseline ChatUniTest (CUT). However, the results show substantial variability, as indicated by large standard deviations. We can also spot that the highest mean for public projects is obtained by the Prompt Engineering technique alone (PO). In contrast, in the case of private projects, the combined implementation (ALL) performed better. Regarding the EDR implementation, it achieves slightly higher coverage than the baseline on average. Coverage in public projects is higher than on private ones, likely because the LLM is unfamiliar with the private dependencies. We also observe that the differences between the default implementation (CUT) and the best implementation in each case (PO in public and ALL in private) is larger for private projects.

Table 2. Mean and standard deviation (subscript) of branch coverage percentage for public and private projects. The highest value for every group of projects is highlighted.

	CUT	***PO***	***EDR***	***ALL***
Public Projects	$41.91_{16.82}$	$\mathbf{50.40}_{14.07}$	$42.39_{14.31}$	$46.53_{15.64}$
Private Projects	$28.66_{10.84}$	$29.60_{18.59}$	$31.75_{13.96}$	$\mathbf{40.18}_{12.04}$

Given the non-deterministic nature of our techniques, we performed multiple independent runs and analyzed the results with statistical tests. First, we applied Wilcoxon signed-rank test to identify significant differences between implementations. Next, we used the Vargha–Delaney $A12$ effect-size measure to determine which implementation accounts for the observed differences.

In Table 3 we show the $A12$ measure results. In addition, values showing statistically significant differences (p-value < 0.0083) according to the Wilcoxon signed-rank test are highlighted in bold. For values greater than 0.5 the row implementation performed better, meanwhile, for values smaller than 0.5 column implementation performed better. As we can see, in public projects there is a statistically significant difference in performance of ChatUniTest using Optimized Prompt (PO) compared to ChatUniTest default implementation (CUT). This difference is in favor of the Prompt Engineering technique, which has a 55.9% probability of achieving higher branch coverage than the default implementation.

When generating test suites for private project methods, the Prompt Engineering and External Dependency Retrieval combination (ALL) obtained statistically significant differences compared to the rest of the implementations. The combination of the two techniques (ALL) showed a better performance than the default implementation in 60.8% of the cases.

Table 3. *A*12 test results for pairwise comparison of implementations in public and private projects based on branch coverage. Statistically significant differences are highlighted in bold.

		PO	***EDR***	***ALL***
Public Projects	***CUT***	**0.440**	0.501	0.476
	PO	–	0.559	0.538
	EDR	–	–	0.480
Private Projects	***CUT***	0.458	0.464	**0.392**
	PO	–	0.502	**0.422**
	EDR	–	–	**0.425**

To better understand the performance differences between the implementations across all methods, we also conducted a Wilcoxon rank-sum test, summarized in Table 4. Note that the threshold for statistically significant differences is set at p-value < 0.0083. A ▲ indicates that the implementation in the row significantly outperformed the one in the column, whereas ▼ indicates that the implementation in the column significantly outperformed the one in the row. In public projects, Prompt Engineering (PO) showed good performance, but it was not as strong as the combined Prompt Engineering and External Dependency Retrieval (ALL) in private projects. The latter achieved significantly better coverage in 44 methods compared to the default ChatUniTest implementation (CUT). In public projects, the External Dependency Retrieval only implementation (EDR) is better than the baseline in 4 methods and worse in 5. In private projects, however, EDR outperformed the baseline in 13 methods and underperformed it in 6.

Table 4. Wilcoxon rank-sum results for all 80 methods in public projects and 181 methods in private projects based on branch coverage. Results are shown relative to the implementations listed in each row.

		PO	***EDR***	***ALL***
Public Projects	***CUT***	▲4 ▼12	▲5 ▼4	▲5 ▼10
	PO	–	▲12 ▼6	▲2 ▼2
	EDR	–	–	▲2 ▼9
Private Projects	***CUT***	▲8 ▼20	▲6 ▼13	▲7 ▼44
	PO	–	▲21 ▼13	▲1 ▼23
	EDR	–	–	▲1 ▼35

Answering RQ1, the Prompt Engineering + External Dependency Retrieval implementation significantly outperforms the default implementation of ChatUniTest, especially on industrial software. While prompt optimization alone obtains improvements on public projects, combining it with external dependency retrieval leads to substantially larger branch coverage gains in propietary meth-

ods. These results demonstrate that enriching an optimized prompt with external dependency code effectively boosts branch coverage.

This improvement is likely because the LLM's training data includes public libraries common to open-source projects, whereas proprietary industrial libraries remain unknown to the model, making additional context essential for generating useful test suites.

4.2 Test Suite Costs

The LLM's API cost is a key factor analyzed in this section. Specifically, we analyze here the input and output tokens required to generate test suites for each method. In Table 5, we observe that implementations using External Dependency Retrieval (*EDR* and *ALL*) use more input tokens compared to the other two implementations. The increase in input tokens is caused by the nature of the context retrieving technique, which has more information in its prompt than the regular ones. In contrast, we obtain significantly higher coverage. In private projects there is also a difference between the average output tokens generated between *ALL* and the rest of the implementations, being this one lower than the rest. Output token consumption is influenced by the number of reparation steps performed by each implementation. On average, private projects require the following number of reparation steps: 2.34 for *CUT*, 1.83 for *PO*, 1.89 for *EDR* and 1.30 for *ALL*.

Table 5. Average input and output tokens consumption per method for all implementations.

	Input				**Output**			
	CUT	*PO*	*EDR*	*ALL*	*CUT*	*PO*	*EDR*	*ALL*
Public Projects	4,841	4,820	14,265	13,665	1,780	1,790	1,726	1,779
Private Projects	10,425	9,918	18,636	16,267	3,090	2,687	2,685	2,190

In Table 6, we show the $A12$ effect size values comparing input and output token counts. Statistically significant differences identified by the Wilcoxon test are highlighted in bold. Significant differences in the number of input tokens were observed, especially when comparing implementations with and without External Dependency Retrieval, across both public and private projects. Specifically, *EDR* and *ALL* consumed more input tokens in 71.9% and 71.7% of their respective comparisons with *CUT*. In contrast, for output tokens in private projects, the tend reverses: implementations that included External Dependency Retrieval in the prompt require significantly fewer tokens than those that do not. The only exception is *EDR*, which did not significantly outperform *PO*. Notably, *ALL* required fewer output tokens than *CUT* in 66.2% of the cases.

Answering RQ2, increasing the amount of context—particularly by incorporating external dependencies through External Dependency Retrieval does

Table 6. *A*12 test results for pairwise comparison of implementations in public and private projects based on token consumption. Statistically significant differences are highlighted in bold.

		Input			Output		
		PO	***EDR***	***ALL***	***PO***	***EDR***	***ALL***
Public Projects	***CUT***	0.489	**0.281**	**0.265**	0.487	0.506	0.489
	PO	–	**0.283**	**0.267**	–	0.524	0.503
	EDR	–	–	0.487	–	–	0.479
Private Projects	***CUT***	0.527	**0.368**	**0.386**	0.581	**0.570**	**0.662**
	PO	–	**0.338**	**0.355**	–	0.508	**0.606**
	EDR	–	–	0.521	–	–	**0.570**

lead to a higher number of input tokens. However, this increase is partially offset by a reduction in output tokens. In those cases, more informative prompts lead to fewer failed attempts and more concise test generation. For instance, in Codestral-2501 paid API (used in this work) and other models such as GPT-4.1 the cost per output token tend to be three times higher than the input token cost. Therefore, assuming that input token cost is 1 unit and output token cost is 3 units, the costs per method are higher in public projects using External Dependency Retrieval (19,443 units for EDR and 19,002 units for ALL) than for default implementation (10,181 units).

In private projects, although ALL still incur higher costs (22,837 units for ALL and 19,695 units for CUT), the combination of Prompt Engineering and External Dependency Retrieval techniques helps mitigate this increase. This is mainly because fewer output tokens are required, as the need for repair steps are reduced. Consequently, the higher number of input tokens (16,267 in ALL and 10,425 in CUT) does not translate into a proportionally higher overall cost, partially compensating the apparent difference.

5 Conclusions and Future Work

This work presented two complementary techniques for enhancing LLM-based test case generation: Prompt Engineering and External Dependency Retrieval. In private projects, the combined implementation (ALL) achieves average branch coverage gains of 11.52% over the baseline (CUT), 10.58% over the prompt-optimized implementation (PO), and 8.43% over the context-enriched implementation about external dependencies (EDR). In addition, the combined implementation (ALL) shows statistically significant improvements over all other variants. In public projects, prompt optimization alone implementation (PO) performs best, although its advantage is significant only when compared to the baseline. This likely reflects that public libraries are more often included in the LLM's training data, whereas private dependencies are not, reducing the added value of explicit context.

Although the ALL implementation increases input tokens from 10,425 to 16,267 (+56%) in industrial projects, this cost is partly compensated by a re-

duction in output tokens. This reduction is mainly due to fewer repair attempts needed to produce valid test cases: 2.34 attempts on average for CUT versus 1.30 for ALL. Since output tokens are typically three times more expensive, generating fewer of them helps to contain the cost. Overall, ALL incurs only a 15.95% higher cost than the baseline while delivering higher branch coverage.

As future work, we plan to perform a more detailed study on the type and amount of external dependency information provided to the LLM. Our goal is to identify the minimal context required to achieve significant coverage gains, which could help further reduce costs. Additionally, given the importance of prompt design, we aim to explore advanced prompt engineering techniques that dynamically adapt prompts based on the specific characteristics of the code under test.

Acknowledgments

This research has been supported by MICIU/AEI/10.13039/501100011033 under projects PLEC2023-010266 (SOFIA), PID2022-142964OA-I00 (EGSVAI) and RED2022-134647-T (AI4Software). It has also been partially funded by European Regional Development Funds (ERDF) and the University of Malaga.

References

1. Chen, Y., Hu, Z., Zhi, C., Han, J., Deng, S., Yin, J.: Chatunitest: A framework for llm-based test generation. In: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering. p. 572–576. FSE 2024, Association for Computing Machinery, New York, NY, USA (2024)
2. Ma, H., Zhang, C., Bian, Y., Liu, L., Zhang, Z., Zhao, P., Zhang, S., Fu, H., Hu, Q., Wu, B.: Fairness-guided few-shot prompting for large language models. In: Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., Levine, S. (eds.) Advances in Neural Information Processing Systems. vol. 36, pp. 43136–43155. Curran Associates, Inc. (2023)
3. Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B.P., Hermann, K., Welleck, S., Yazdanbakhsh, A., Clark, P.: Self-refine: Iterative refinement with self-feedback. In: Thirty-seventh Conference on Neural Information Processing Systems (2023)
4. Pacheco, C., Ernst, M.: Randoop: Feedback-directed random testing for java. pp. 815–816 (2007)
5. Schweikl, S., Fraser, G., Arcuri, A.: Evosuite at the sbst 2022 tool competition. In: Proceedings of the 15th Workshop on Search-Based Software Testing. p. 33–34. SBST '22, Association for Computing Machinery, New York, NY, USA (2023)
6. Tufano, M., Drain, D., Svyatkovskiy, A., Deng, S.K., Sundaresan, N.: Unit test case generation with transformers. CoRR **abs/2009.05617** (2020)
7. Vargha, A., Delaney, H.D.: A critique and improvement of the "cl" common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics **25**(2), 101–132 (2000)

8. Wang, Z., Liu, K., Li, G., Jin, Z.: Hits: High-coverage llm-based unit test generation via method slicing. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. p. 1258–1268. ASE '24, Association for Computing Machinery, New York, NY, USA (2024)
9. Yuan, Z., Liu, M., Ding, S., Wang, K., Chen, Y., Peng, X., Lou, Y.: Evaluating and improving chatgpt for unit test generation. Proc. ACM Softw. Eng. **1**(FSE) (Jul 2024)