Enhancing LLM's Ability to Generate More Repository-Aware Unit Tests Through Precise Context Injection

Xin Yin[1] Chao Ni[1*] Xinrui Li[1] Liushan Chen[2] Guojun Ma[2*] Xiaohu Yang[1]
[1]*The State Key Laboratory of Blockchain and Data Security, Zhejiang University*,
{xyin, chaoni, lixinrui, yangxh}@zju.edu.cn
[2]*ByteDance*, {chenliushan, maguojun}@bytedance.com

***Abstract*—Recently, Large Language Models (LLMs) have gained attention for their ability to handle a broad range of tasks, including unit test generation. Despite their success, LLMs may exhibit hallucinations when generating unit tests for focal methods or functions due to their lack of awareness regarding the project's global context. While many studies have explored the role of context, they often extract fixed patterns of context for different models and focal methods, which may not be suitable for all generation processes (e.g., excessive irrelevant context could lead to redundancy, preventing the model from focusing on essential information).**

To overcome this limitation, we propose RATester, which integrates language servers to provide dynamic definition lookup to assist the LLM. When RATester encounters an unfamiliar identifier, it first leverages language servers (e.g., Gopls) to fetch relevant definitions and documentation comments, and then uses this global knowledge to guide the LLM. We evaluate the effectiveness and efficiency of RATester by constructing a new Golang dataset from real-world projects. On our Golang dataset, RATester achieves an average line coverage of 26.25%, representing an improvement of 9.10% to 165.69% over the baselines. In mutation testing, RATester shows superior performance by successfully killing 18 to 147 more mutants than the baselines. Additionally, our model-agnostic and generalizability analysis confirms RATester's effectiveness across different models, programming languages, and model scales, validating its broad applicability.

***Index Terms*—Unit Test Generation, Large Language Model, Precise Context**

I. Introduction

Unit testing plays a critical role in software maintenance by enabling developers to identify defects and errors early in the development process, thereby ensuring software system quality. This not only helps reduce overall product costs but also enhances developer productivity [1]–[3]. Despite its significance, manually writing high-quality unit tests remains both challenging and time-consuming.

To address this challenge, researchers are increasingly exploring Large Language Models (LLMs) for automated unit test generation. These LLMs can generate unit tests directly from contextual information, reducing reliance on task-specific datasets by leveraging their extensive pre-training on large-scale open-source code repositories. Recent studies [4], [5] have adopted ChatGPT to generate unit tests based on focal methods. However, despite these advancements, LLMs can still exhibit hallucinations when generating unit tests due to their limited awareness of the project's global context. These hallucinations typically manifest as calling non-existent methods, assigning incorrect parameters and return values (e.g., mismatched parameter types or incorrect parameter counts). To overcome this limitation, many studies have explored context extraction techniques to reduce hallucinations. ChatUniTest [6] introduces an LLM-based framework that enhances unit test generation through an adaptive focal context mechanism, effectively capturing relevant context within prompts. It also employs a "Generation-Validation-Repair" process to fix errors in the generated tests. Subsequently, researchers [5], [7], [8] have explored the roles of focal context and dependency context. These approaches utilize the focal method and class to extract context, including: (1) focal class signatures; (2) signatures of other methods and fields within the class; (3) signatures of dependent classes; and (4) signatures of dependent methods and fields in dependent classes. However, these extraction patterns suffer from two key issues. First, when essential context cannot be extracted based on classes and dependencies, they may overlook important context: the definitions and comments for unfamiliar identifiers utilized during unit test generation (e.g., the Context struct in Fig. 1). Second, they may introduce redundant context by extracting unused information, such as unnecessary dependency definitions (e.g., the BasePath function in Fig. 1). This excessive, irrelevant context can hinder the model's ability to focus on essential information.

In practical development scenarios, developers typically possess comprehensive familiarity with the methods, functions, and data structures within their working packages. Moreover, Integrated Development Environment (IDE) tools and language servers provide real-time assistance such as function call information and identifier descriptions through "hovering" over identifiers, enabling developers to write more accurate and contextually appropriate unit tests. Motivated by this observation, we aim to equip LLMs with projects' global knowledge comparable to that of human developers by introducing RATester, which enhances LLMs' capability to generate repository-aware unit tests through precise context injection. To provide LLMs with precise knowledge, we

*Corresponding authors.
Chao Ni is also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00058

integrate language servers that enable definition lookup and contextual information retrieval. When RATester encounters unfamiliar identifiers (e.g., method names and struct names), it proactively queries the language server to retrieve relevant definitions and documentation comments. RATester emulates the developer workflow by enabling LLMs to dynamically fetch precise, repository-aware context through "hovering" over identifiers during generation. By continuously leveraging language server capabilities, RATester progressively builds the LLM's comprehensive project knowledge, thereby reducing hallucinations and enhancing unit test generation effectiveness.

We construct a comprehensive evaluation dataset comprising eight highly-starred GitHub projects (ranging from 29.7k to 85.5k stars): beego, echo, fiber, frp, gin, hugo, nps, and traefik. To assess the effectiveness and efficiency of RATester, we conduct comparative evaluations against eight approaches spanning three categories: one traditional approach (NxtUnit [9]), one learning-based approach (UniTester [10]), three LLM-based approaches (ChatUniTest [6], ChatTester [5], and SymPrompt [8]), and three foundational LLMs (CodeLlama [11], DeepSeek-Coder [12], and Magicoder [13]). Experimental results demonstrate the superiority of RATester across all evaluation metrics. Specifically, RATester achieves an average line coverage of 26.25%, representing improvements ranging from 9.10% to 165.69% over baselines. Moreover, RATester achieves the highest performance in mutation testing, successfully killing 18 to 147 more mutants compared to baselines.

In summary, the key contributions of this paper include:

A. Novel LLM-based Framework: We present RATester, an LLM-based framework for unit test generation that does not rely on task-specific training datasets. Our results demonstrate that this framework can outperform existing approaches, achieving superior performance in unit test generation.

B. Repository-Aware Tester: We introduce RATester, which utilizes language servers to enhance the LLM's global knowledge of the project. By proactively fetching definitions and documentation comments for unfamiliar information, RATester reduces hallucinations during unit test generation.

C. Comprehensive Evaluation: (1) We conduct studies on the effectiveness and efficiency of RATester by collecting a new Golang dataset from real-world projects. (2) We evaluate RATester not only using compile rate and line coverage metrics but also assess its capabilities in mutation testing. (3) The replication of this paper is publicly available [14].

II. Motivation

A. A Motivation Example

Fig. 1 shows a focal method named "PATCH" along with the unit tests generated by DeepSeek-Coder for a Golang project named gin. The upper right corner of Fig. 1 illustrates how DeepSeek-Coder (using imprecise context) generates a unit test for the focal method without precise knowledge of the project. The unit test "TestPATCH" verifies whether the server can correctly handle an HTTP PATCH request sent to the "/patch" path and returns the expected response status code of "http.StatusOK" and the response body "Hello, World". The test creates a route instance, defines a handler for the PATCH request, and then uses the "httptest" package to simulate the request and capture the response, ultimately checking whether the response's status code and body meet expectations. However, in the fourth line, this unit test encounters a compilation error: "c.String(http.StatusOK, 'Hello, World') (no value) used as value", preventing the test from compiling. This issue arises because DeepSeek-Coder (using imprecise context) lacks sufficient knowledge of the project and does not know that the "String" method within the "Context" struct does not return a value, leading to hallucinations during inference.

Observation. Since existing approaches utilize focal methods and classes to extract context, there is a certain degree of knowledge omission and redundancy in the context provided to LLMs, meaning it may not be the precise context that LLMs truly require during unit test generation. Given the limited input capacity of LLMs, it is impractical to feed all available context into the model. Excessive irrelevant context can introduce redundancy, preventing the model from focusing on essential information, which may result in LLMs exhibiting hallucinations. These hallucinations may include invoking non-existent methods, setting incorrect parameters and return values (e.g., parameter type mismatches or incorrect number of parameters). We should provide models with precise, necessary context for generation, rather than fixed context selections, to minimize the interference of redundant information.

Focal Method

```
func (group *RouterGroup) PATCH(relativePath string,
      handlers ...HandlerFunc) IRoutes {
   return group.handle(http.MethodPatch, relativePath, handlers)
}
```

Imprecise Context (Existing Approaches' Context)

```
type RouterGroup struct {
   Handlers  HandlersChain
   ......
}
func (group *RouterGroup) BasePath() string
func (group *RouterGroup) returnObj() IRoutes
......
```

Precise Context

```
type Context struct {
   writermem  responseWriter
   ......
}

func (c *Context) Abort()
......
func (c *Context) String(code int, format string, values ...any)
......
```

Unit Test Generated Using Imprecise Context

```
func TestPATCH(t *testing.T) {
   router := New()

   router.PATCH("/patch", func(c *Context) error {
      return c.String(http.StatusOK, "Hello, World")
   })
   req := httptest.NewRequest(http.MethodPatch, "/patch", nil)
   rec := httptest.NewRecorder()
   router.ServeHTTP(rec, req)
   ......
}
```

Unit Test Generated Using Precise Context

```
func TestPATCH(t *testing.T) {
   router := New()

   router.PATCH("/patch", func(c *Context) {
      c.String(200, "PATCH method on /patch")
   })
   req, _ := http.NewRequest("PATCH", "/patch", nil)
   w := httptest.NewRecorder()
   router.ServeHTTP(w, req)
   ......
}
```

Fig. 1: A focal method along with the unit tests generated by DeepSeek-Coder for a project named *gin*

B. Key Idea: Proactively Fetch Precise Context

Based on the above observation, we propose RATester, a repository-aware unit test generation framework that emulates the developer workflow by enabling LLMs to dynamically fetch precise context through "hovering" over identifiers during unit test generation. Unlike existing approaches that rely on fixed context extraction patterns, RATester leverages the language servers to proactively retrieve definitions and documentation comments for unfamiliar structs, methods, functions, and other identifiers. This targeted context acquisition ensures that LLMs receive exactly the information they need, thereby effectively mitigating hallucinations. For example, as illustrated on the right side of Fig. 1, when RATester encounters the unfamiliar method "Context", it proactively queries Gopls to fetch specific

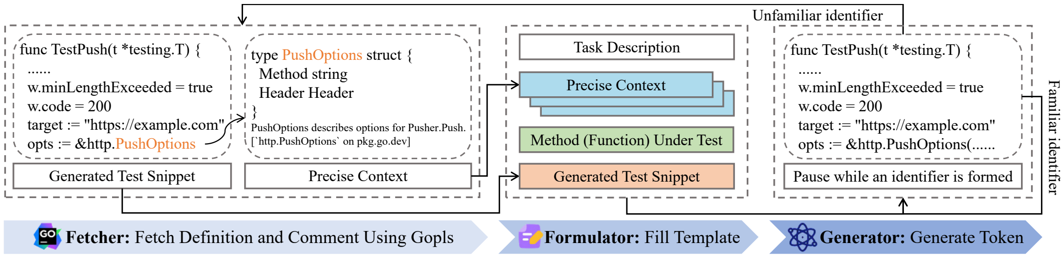

Fig. 2: Overview of RATester (using Golang as an example)

definitions and documentation comments, preventing erroneous usage such as the error shown in the motivating example. By continuously leveraging language server capabilities throughout the unit test generation, RATester progressively builds the LLM's comprehensive project understanding, resulting in more accurate and repository-aware unit tests.

III. Our Approach: RATester

A. Overview

RATester simulates developers' workflow by enabling LLMs to dynamically fetch precise, repository-aware context via "hovering" over identifiers during unit test generation, addressing imprecise context issues in previous works. As shown in Fig. 2, RATester consists of three components: Fetcher, Formulator, and Generator. Given the focal method/function that needs to be tested, each component plays a distinct role:

- **Fetcher** fetches the definition and documentation comment of unfamiliar identifiers using the language server (e.g., Gopls).
- **Formulator** fills the fetched precise context and the test snippet with newly generated identifiers into the prompt template, and then formulates them as input for the generator.
- **Generator** leverages the formulated input to perform the unit test generation task.

B. Fetcher

When generating unit tests for focal methods or functions, LLMs often produce hallucinations, which can manifest as calls to non-existent methods, as well as incorrect parameter assignments and return values, such as mismatched parameter types or an improper number of parameters. In contrast, human testers typically possess a strong understanding of the various methods, functions, and structs within the package during development. Additionally, IDE tools and language servers provide essential support by offering information on function calls and identifier descriptions, facilitating the creation of accurate code. Consequently, human testers frequently leverage insights from these tools to enhance their global knowledge while crafting unit tests, ultimately reducing the likelihood of erroneous test cases.

To replicate this developer workflow, RATester serves as an intelligent context fetcher that utilizes language servers to equip LLMs with precise, repository-aware knowledge comparable to that of human developers. We use Golang as an example. RATester integrates Gopls [15], the official Go language server, which facilitates interactions with editors such as Visual Studio Code. Gopls can assess the context of identifiers in the generated unit test snippet, which includes function definitions, method definitions, struct definitions, and various parameter definitions and their corresponding documentation comments. This targeted information retrieval enables RATester to enhance the LLM's repository understanding progressively, thereby reducing hallucinations and improving test generation accuracy.

In the initial stage, RATester actively queries Gopls for the definitions and documentation comments of the receiver type, parameter types, and return type of the focal method/function. All retrieved information is then incorporated into the prompt template. During the continuous stage, whenever the LLM generates an unfamiliar identifier (e.g., new function names or struct names), RATester proactively utilizes Gopls to verify the identifier's existence within the current package and fetches its corresponding definitions and documentation comments. The Formulator then incorporates this fetched information into the prompt template (refer to Section III-C for details). By leveraging Gopls to dynamically fetch context at both stages, RATester enables the LLM to develop a comprehensive understanding of the project's codebase. This approach mirrors how human developers utilize IDE tools (e.g., hovering over identifiers) to access definitions and documentation for unfamiliar code elements during development process.

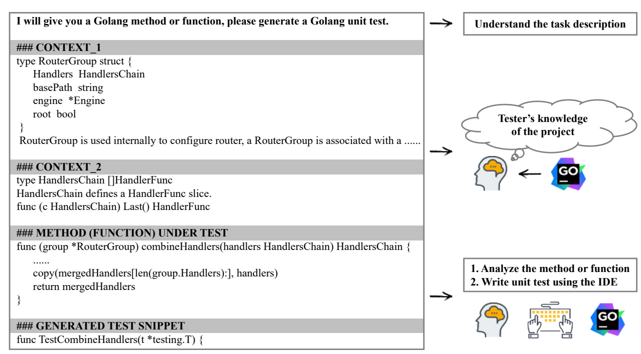

Fig. 3: An example of prompt for the unit test generation

C. Formulator

In both the initial stage and the continuous stage, the formulator fills the fetched precise context and the test snippet into the prompt template. As shown in Fig. 3, this prompt template consists of four main parts:

- **Task Description.** RATester provides the LLM with the description constructed as *"I will give you a Golang method or function, please generate a Golang unit test"*. This part aids the LLM in understanding the task description, simulating the process by which a human tester comprehends the objectives of the task.

- **Precise Context.** RATester provides the fetched definitions and documentation comments to LLM. The precise context continuously expands as the generation process progresses, enhancing the LLM's global knowledge of the project. This part simulates the global project knowledge that human testers possess with the assistance of IDE tools.
- **Method (Function) Under Test.** RATester provides the focal method or function to LLM. We also prefix the focal method or function with *"### METHOD (FUNCTION) UNDER TEST"* to directly indicate LLM about the context of the method or function. This part simulates the scenario in which human testers review the method or function being tested.
- **Generated Test Snippet**. RATester provides the LLM with the unit test generated in the previous round, along with a new identifier. In the initial stage, the generated test snippet is explicitly set as *"func Test{name}(t *testing.T)"*. As this part continues to expand, it simulates the iterative process of human testers writing unit tests.

D. Generator

The generator leverages results returned from the formulator and performs the tasks of unit test generation accordingly. It continually generates tokens until a complete identifier is formed. If the generated identifier is not found in the precise context, RATester uses the Gopls to fetch the definition and documentation comments, which are then filled into the prompt template for the next generation step. By continuously leveraging Gopls to fetch the precise context, the LLM acquires sufficient global knowledge, thereby reducing the likelihood of hallucinations.

IV. Experimental Design

In this section, we first present our collected dataset and then introduce the baselines. Following that, we describe the performance metrics as well as the experimental setting.

TABLE I: The statistics of the constructed dataset for Golang

Project	Star	Focal Method and Function (#)	Line Coverage of Unit Tests (%)
beego	31.5k	2,688	38.78%
echo	29.7k	419	93.58%
fiber	33.6k	765	85.46%
frp	85.5k	864	2.59%
gin	78.5k	449	95.53%
hugo	75.4k	3,829	76.54%
nps	30.6k	455	0.51%
traefik	50.9k	1,726	58.95%

A. Dataset Construction

We construct a dataset to evaluate RATester for Golang by selecting the top 15 most-starred Golang projects from GitHub, filtered by runtime compatibility and requiring > 100 focal methods/functions. This process yields eight diverse projects spanning multiple domains (i.e., Web Frameworks, Networking & Proxy, and Content Management) with stars ranging from 29.7k to 85.5k: beego, echo, fiber, frp, gin, hugo, nps, and traefik. Since RATester focuses on generating unit tests for focal methods and functions, we extract all methods and functions from each project. The detailed statistics are shown in Table I. In addition to the star count and the number of successfully extracted focal methods and functions, we also run all unit tests within the projects and show the line coverage.

B. Baselines

To comprehensively evaluate the effectiveness of RATester, we carefully select baselines based on three key criteria: approach category, Golang-specific design capabilities, and reproducibility. Accordingly, we consider eight representative baselines spanning different methodological paradigms: one traditional approach, one learning-based approach, and six LLM-based approaches.

Traditional Approach. To present the traditional approach, we employ NxtUnit [9], an automatic unit test generation tool for Golang that leverages random testing and is particularly suited for microservice architectures. It offers three types of interfaces: an IDE plugin, a CLI, and a web-based platform. NxtUnit's random-based strategy allows it to quickly generate unit tests, making it ideal for smoke testing and rapid quality feedback. However, NxtUnit may sometimes fail to generate test cases due to issues like compilation errors or test crashes. As a result, NxtUnit only provides test cases that can be executed successfully. In our evaluation, we use the default settings from the original paper.

Learning-based Approach. To present the learning-based approach, we utilize the transformer-based generation model, UniTester [10]. This model is trained on the UniTSyn dataset and is capable of synthesizing unit tests for programs in multiple languages, including Golang. As the published code for UniTester lacks the model checkpoint, we retrain UniTester following the settings described in the paper and using the UniTSyn dataset. To prevent data leakage, we exclude projects from the training set that overlap with those in our dataset.

LLM-based Approach. To represent the LLM-based approach, we utilize ChatUniTest [6], ChatTester [5], SymPrompt [7], and three LLMs to generate unit tests for each focal method and function without fine-tuning. The models we select are recently released: CodeLlama [11], DeepSeek-Coder [12], and Magicoder [13]. For ChatUniTest, ChatTester, and SymPrompt, we follow the original settings described in their respective papers, specifically setting the repair attempts to 5 for ChatUniTest and the accumulated number of invalid refinements to 3 for ChatTester.

C. Evaluation Metrics

To evaluate the performance of RATester and baseline approaches, we use Compile Rate and Line Coverage:

Compile Rate represents the proportion of test cases that can be successfully compiled and executed out of the total number generated. A higher compile rate reflects better quality and reliability in the generated test cases.

Line Coverage quantifies the percentage of code lines executed by the test cases, offering insights into the effectiveness of

TABLE II: RQ-1: RATester *vs.* Baselines across different Golang projects in compile rate and line coverage

Projects	Compile Rate						Line Coverage					
	NxtUnit	UniTester	ChatUniTest	ChatTester	SymPrompt	RATester	NxtUnit	UniTester	ChatUniTest	ChatTester	SymPrompt	RATester
beego	-	31.87%	56.38%	53.14%	58.36%	68.75%	20.51%	12.71%	27.85%	27.02%	28.61%	31.91%
echo	-	22.36%	39.65%	39.43%	40.39%	45.58%	10.77%	9.89%	23.09%	21.53%	23.48%	24.50%
fiber	-	21.22%	42.44%	43.21%	40.98%	59.61%	20.24%	9.33%	19.33%	20.25%	23.17%	26.31%
frp	-	28.57%	52.35%	50.44%	47.79%	66.90%	12.84%	8.62%	13.20%	11.97%	14.01%	14.43%
gin	-	32.91%	63.56%	63.68%	59.02%	69.49%	21.92%	11.38%	53.92%	52.33%	55.27%	58.09%
hugo	-	24.87%	48.76%	43.78%	42.11%	61.51%	12.34%	9.57%	18.78%	20.55%	21.96%	25.02%
nps	-	16.67%	53.11%	54.67%	57.20%	64.84%	16.48%	7.49%	12.66%	14.61%	14.09%	16.82%
traefik	-	18.44%	46.25%	43.80%	43.64%	58.05%	10.45%	10.01%	11.69%	10.97%	11.86%	12.92%
Average	-	24.61%	50.31%	49.02%	48.69%	61.84%	15.69%	9.88%	22.57%	22.40%	24.06%	26.25%

the tests in covering the code. A higher line coverage indicates that a larger portion of the code lines is being tested.

While Compile Rate and Line Coverage are valuable, they do not fully assess test quality. To provide a more comprehensive evaluation of the unit tests generated by RATester, we also employ mutation testing. We use Gremlins [16] to introduce mutations into the projects and evaluate the number of mutants killed by the unit tests, along with mutator coverage.

D. Implementation Details

We develop the unit test generation in Python, utilizing PyTorch [17] implementations of LLMs (i.e., CodeLlama 7B, DeepSeek-Coder 6.7B, and Magicoder 6.7B). We use the Hugging Face API [18] to load the model weights and generate outputs. For the baseline comparisons, we directly use the settings provided in their original papers to generate unit tests. Considering both the performance improvements and the associated generation costs, we generate one unit test for each focal method (refer to Section V-C for more details) and test them using the test command. Our evaluation is conducted on a 32-core workstation equipped with an Intel(R) Xeon(R) Platinum 8358P CPU 2.60GHz, 2TB RAM, and 8×NVIDIA A800 80GB GPU, running Ubuntu 20.04.6 LTS.

V. Experimental Results

To investigate the effectiveness and efficiency of RATester on unit test generation, our experiments focus on the following three research questions:

- **RQ-1 Effectiveness Comparison.** *How does the effectiveness of RATester compare with the baselines?*
- **RQ-2 Model-Agnostic Analysis.** *What are the model-agnostic capabilities of RATester?*
- **RQ-3 Efficiency Comparison.** *How does the efficiency of RATester compare with the baselines?*
- **RQ-4 Generalizability Analysis.** *How well does RATester generalize across different programming languages and model scales?*

A. RQ-1: Effectiveness of RATester

Objective. To reduce the hallucination issues that LLMs experience during unit test generation (e.g., invoking non-existent methods and setting incorrect parameters and return values), we propose the RATester approach. This approach utilizes the definition lookup feature provided by language servers (e.g., Gopls) to dynamically fetch relevant context during the generation. By supplying LLMs with more project-specific knowledge, we aim to minimize hallucinations. In this RQ, our objective is to investigate whether RATester outperforms previous baselines in terms of effectiveness.

Experimental Design. We consider five baselines: NxtUnit [9], UniTester [10], ChatUniTest [6], ChatTester [5], and SymPrompt [7]. To facilitate a fair comparison, we employ DeepSeek-Coder as the backend model for RATester, ChatUniTest, ChatTester, and SymPrompt.

For a comprehensive performance comparison between the baselines and RATester, we conduct two distinct experiments across eight real-world projects. The first experiment involves executing all generated unit tests, recording the compile rate and line coverage. In the second experiment, we extend our evaluation with mutation testing, using Gremlins [16] to mutate the projects and measure the number of mutants killed by the generated unit tests, along with the mutator coverage. This experiment demonstrates the effectiveness of unit tests generated by RATester in detecting unknown defects.

Results. We discuss the results from the aspects of compile rate, line coverage, and mutation testing, respectively.

Effectiveness of RATester in Compile Rate and Line Coverage. Table II shows the compile rate and line coverage of the generated unit Tests. We observe that RATester consistently outperforms the baselines across all projects. Specifically, the compile rate of unit tests generated by RATester significantly improves from a range of 16.67%–63.68% to 45.58%–69.49%. Note that NxtUnit only provides test cases that can be executed successfully. Therefore, we do not record the compile rate for the test cases generated by NxtUnit.

In addition to the compile rate, RATester demonstrates a significant enhancement in line coverage. Across all evaluated projects, RATester increases the line coverage from a range of 7.49%–55.27% to 12.92%–58.09%. This results in an average improvement of 9.10%–165.69% when compared to the baseline approaches. Such a substantial increase in line coverage indicates that RATester is more effective in generating comprehensive unit tests, thereby enhancing the overall robustness of the tested software.

In Table IV, we evaluate the line coverage achieved by combining the original unit tests with the generated unit tests. The "Origin+{Approach}" column displays the total coverage achieved by combining the original tests with unit tests generated by different approaches. Overall, both RATester and the baselines successfully enhance the line coverage of

TABLE III: RQ-1: RATester *vs.* Baselines across different Golang projects in mutation testing

Projects	Mutants (#)	Killed Mutants (#)						Mutator Coverage (%)					
		NxtUnit	UniTester	ChatUniTest	ChatTester	SymPrompt	RATester	NxtUnit	UniTester	ChatUniTest	ChatTester	SymPrompt	RATester
beego	4,573	468	138	425	412	429	473	17.93%	8.19%	18.43%	17.96%	19.05%	19.91%
echo	922	49	26	90	81	85	91	15.59%	12.21%	16.39%	15.79%	16.24%	16.59%
fiber	1,554	153	80	141	139	148	160	15.24%	9.36%	15.93%	15.61%	16.80%	17.99%
frp	1,538	84	57	83	87	93	99	6.95%	2.68%	6.88%	6.92%	7.26%	7.76%
gin	885	127	61	153	131	157	164	18.21%	12.73%	25.76%	22.10%	25.89%	26.05%
hugo	5,882	542	339	620	601	645	670	9.27%	7.75%	11.07%	10.71%	11.13%	11.36%
nps	1,369	52	41	50	44	48	56	7.68%	4.29%	7.56%	7.12%	7.79%	8.36%
traefik	4,331	269	109	264	265	277	308	7.35%	5.88%	7.67%	7.48%	8.22%	8.84%
Average	2,632	218	106	228	220	235	253	12.28%	7.89%	13.71%	12.96%	14.05%	14.61%

the original unit tests; however, RATester demonstrates a more significant improvement. Specifically, the line coverage increases substantially from a range of 0.51%–95.53% to 14.46%–95.57%. In addition, the average line coverage also shows a notable improvement, rising from 56.49% to 60.98%. This indicates that RATester not only enhances line coverage but also serves to complement human-written unit tests, thereby contributing to better overall software quality.

TABLE IV: RQ-1: The line coverage achieved by combining the original unit tests with those generated by RATester and the baselines

Projects	Origin+ NxtUnit	Origin+ UniTester	Origin+ ChatUniTest	Origin+ ChatTester	Origin+ SymPrompt	Origin+ RATester
beego	38.79%	38.78%	40.93%	39.86%	41.37%	41.82%
echo	93.58%	93.59%	93.68%	93.72%	93.77%	93.96%
fiber	86.15%	85.56%	86.35%	86.18%	86.37%	86.59%
frp	13.51%	9.31%	14.01%	13.91%	14.20%	14.46%
gin	95.54%	95.54%	95.56%	95.54%	95.56%	95.57%
hugo	76.58%	76.60%	76.77%	76.83%	76.90%	77.20%
nps	16.50%	7.56%	12.29%	12.72%	14.50%	16.83%
traefik	59.13%	58.99%	59.93%	60.08%	60.72%	61.39%
Average	59.97%	58.24%	59.94%	59.86%	60.42%	60.98%

Effectiveness of RATester in Mutation Testing. Table III presents the results of the study. For each project listed in column 1, the table details the number of generated mutants (column 2), the number of killed mutants by each approach (columns 3-8), and the mutator coverage percentages (columns 9-14). As shown in Table III, we find that the unit tests generated by RATester not only kill the highest number of mutants but also achieve the best mutator coverage across all evaluated projects. For instance, in the "gin" project, a total of 885 mutants are generated. NxtUnit, UniTester, ChatUniTest, ChatTester, and SymPrompt kill 127, 61, 153, 131, and 151 mutants, respectively. In contrast, RATester achieves an impressive 164 mutants killed, significantly surpassing the performance of the baselines. Furthermore, RATester achieves a mutator coverage of 26.05%, which is notably higher than NxtUnit's 18.21%, UniTester's 12.73%, ChatUniTest's 25.76%, ChatTester's 22.10%, and SymPrompt's 25.89%.

Answer to RQ-1*: RATester significantly outperforms baselines in enhancing compile rate and line coverage, improving from 16.67%–63.68% to 45.58%–69.49% and from 7.49%–55.27% to 12.92%–58.09%, respectively. It also surpasses other approaches in mutation testing, demonstrating its effectiveness in boosting software testing and quality.*

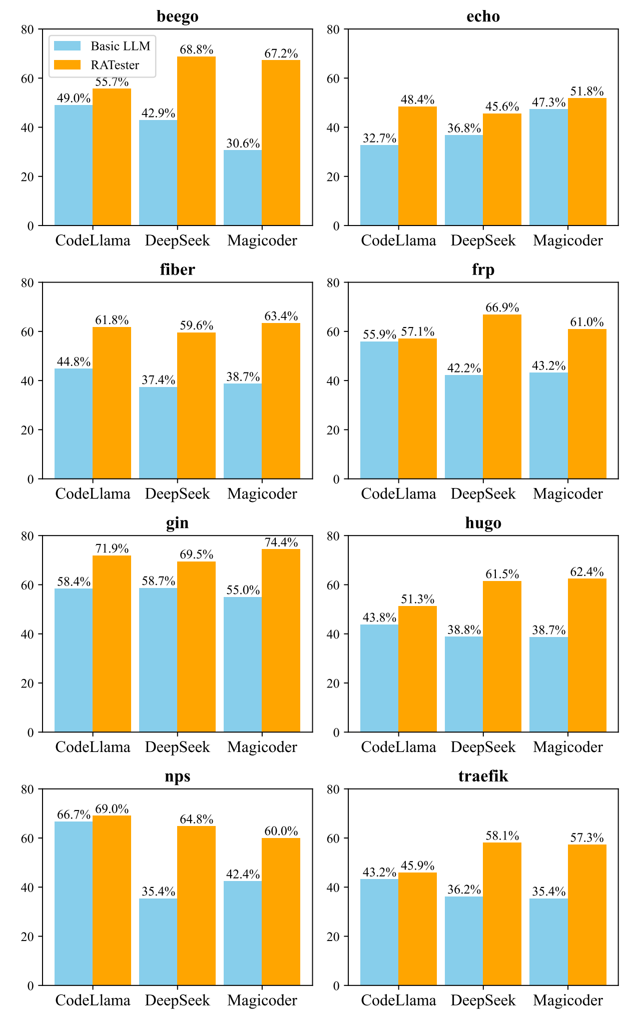

Fig. 4: RQ-2: RATester *vs.* Basic LLMs across different Golang projects in compile rate

B. RQ-2: Model-Agnostic Capabilities of RATester

Objective. In RQ-1, we use DeepSeek-Coder as the backbone model to evaluate the effectiveness of RATester compared to the baselines. The results demonstrate that RATester outperforms existing approaches and shows promising performance in generating unit test cases. In this RQ, we extend our analysis to examine the model-agnostic effectiveness of RATester, specifically assessing whether RATester maintains its effectiveness

TABLE V: RQ-2: RATester *vs.* Basic LLMs across different Golang projects in line coverage

Projects	Basic LLM			RATester		
	CodeLlama	**DeepSeek**	**Magicoder**	**CodeLlama**	**DeepSeek**	**Magicoder**
beego	20.95%	22.75%	22.35%	26.04% (↑24.3%)	31.91% (↑40.3%)	34.71% (↑55.3%)
echo	17.65%	21.08%	24.21%	26.35% (↑49.3%)	24.50% (↑16.2%)	27.55% (↑13.8%)
fiber	14.94%	17.64%	14.82%	20.72% (↑38.7%)	26.31% (↑49.1%)	16.41% (↑10.7%)
frp	11.68%	11.00%	12.95%	14.11% (↑20.8%)	14.43% (↑31.2%)	18.54% (↑43.2%)
gin	43.53%	45.35%	42.67%	48.23% (↑10.8%)	58.09% (↑28.1%)	47.13% (↑10.5%)
hugo	16.10%	16.87%	16.02%	19.77% (↑22.8%)	25.02% (↑48.3%)	18.92% (↑18.1%)
nps	11.83%	10.22%	11.33%	13.12% (↑10.9%)	16.82% (↑64.6%)	14.93% (↑31.8%)
traefik	13.72%	11.68%	15.26%	15.64% (↑14.0%)	12.92% (↑10.6%)	18.62% (↑22.0%)
Average	18.80%	19.57%	19.95%	23.00% (↑22.3%)	26.25% (↑34.1%)	24.60% (↑23.3%)

when applied to different models.

Experimental Design. In addition to DeepSeek-Coder, we utilize two state-of-the-art open-source LLMs to investigate whether RATester remains effective across different models. Specifically, the additional models are (1) CodeLlama [11] and (2) Magicoder [13]. We follow the same experimental setup outlined in Section IV and compare the performance of each model in terms of compile rate, line coverage, and mutation testing. To ensure consistency, we maintain identical experimental conditions across all basic LLMs and their corresponding RATester implementations.

Results. We discuss the model-agnostic capabilities of RATester from the aspects of compile rate, line coverage, and mutation testing, respectively.

Model-agnostic capabilities of RATester in compile rate. Fig. 4 shows the compile rate of unit tests generated by RATester compared to basic LLMs. We find that RATester significantly improves the performance of these basic LLMs. For instance, in the fiber project, RATester increases the compile rate from 44.8% to 61.8% for CodeLlama, from 37.4% to 59.6% for DeepSeek-Coder, and from 38.7% to 63.4% for Magicoder. Overall, RATester raises the compile rate of basic LLMs from a range of 30.6%-66.7% to 45.6%-74.4%, making more unit tests usable by developers during testing. This not only demonstrates the effectiveness of the RATester approach but also highlights its universal applicability. It is designed to be model-agnostic, meaning it can adapt to various LLMs, further emphasizing its flexibility and universality.

Model-agnostic capabilities of RATester in line coverage. The left side of Table V shows the line coverage results of basic LLMs, while the right side presents the results of RATester using different backbone models. We calculate not only the line coverage for RATester with different backbone models but also the improvement percentages relative to basic LLMs. As shown in the table, the RATester approach increases the overall line coverage of unit tests generated by basic LLMs across different projects. For example, in the beego project, RATester improves CodeLlama's coverage by 24.3% (from 20.95% to 26.04%), DeepSeek-Coder's coverage by 40.3% (from 22.75% to 31.91%), and Magicoder's coverage by 55.3% (from 22.35% to 34.71%). Overall, RATester enhances the performance of basic LLMs, raising their average line coverage from a range of 18.80%-19.95% to 23.00%-26.25%, with relative improvements ranging from 22.3% to 34.1%. This aligns with the motivation behind our method's design: providing LLMs with more effective context (such as definitions of called methods) enhances the model's ability to generate unit tests and subsequently increases overall line coverage.

Model-agnostic capabilities of RATester in mutation testing. Table VI presents the results of RATester in using different backbone models in terms of mutation testing. From the results, we find that: **(1) Performance Variation Across Backbone Models:** There are significant performance differences among the basic LLMs, which directly impact the effectiveness of RATester. Among the various backbone models tested, DeepSeek-Coder demonstrates superior performance, leading to the highest effectiveness when RATester utilizes DeepSeek-Coder as its backbone model. **(2) Enhancement Across All Models:** RATester consistently improves the performance of all three basic LLMs utilized in this study. For instance, in the hugo project, Magicoder kills only 522 mutants. In contrast, RATester using Magicoder successfully kills 583 mutants, showcasing a clear enhancement in defect detection capabilities. **(3) Overall Improvement in Defect Detection:** Across all three models tested, RATester exhibits a significant advantage, killing between 316 and 522 more mutants compared to the basic LLMs. This indicates that RATester not only leverages the strengths of the underlying models but also enhances their overall effectiveness in detecting defects.

Answer to RQ-2*: The basic LLMs have limited capabilities in generating unit tests, while RATester enhances these capabilities through appropriate adaptations. Overall, RATester significantly outperforms the basic LLMs in compile rate, line coverage, and mutation testing, demonstrating its adaptability and improved effectiveness.*

C. RQ-3: Efficiency of RATester

Objective. In this RQ, we aim to study the efficiency of RATester. We conduct a comprehensive experiment to evaluate its efficiency. Our focus is on the time and tokens required to generate unit tests, as well as the impact of the number of candidate unit tests generated for each focal method/function.

Experimental Design. We begin by investigating the total time and tokens required to generate test cases for all eight Golang projects. We use the baselines (i.e., NxtUnit, UniTester, ChatUniTest, CodeLlama, DeepSeek-Coder, and Magicoder) to compare the total generation time of RATester across all

TABLE VI: RQ-2: RATester *vs.* Basic LLMs across different Golang projects in mutation testing

Projects	Mutants (#)	Basic LLM			RATester		
		CodeLlama	DeepSeek	Magicoder	CodeLlama	DeepSeek	Magicoder
beego	4,573	209	381	224	388 (+179)	473 (+92)	524 (+300)
echo	922	42	48	50	48 (+6)	91 (+43)	61 (+11)
fiber	1,554	126	131	115	131 (+5)	160 (+29)	124 (+9)
frp	1,538	67	70	69	80 (+13)	99 (+29)	113 (+44)
gin	885	59	135	86	78 (+19)	164 (+29)	115 (+29)
hugo	5,882	428	615	522	448 (+20)	670 (+55)	583 (+61)
nps	1,369	34	50	46	46 (+12)	56 (+6)	56 (+10)
traefik	4,331	145	241	156	207 (+62)	308 (+67)	214 (+58)
Sum	21,054	1,110	1,671	1,268	1,426 (+316)	2,021 (+350)	1,790 (+522)

projects. We choose ChatUniTest as a representative of the LLM-based approaches because of its good performance in both compile rate and line coverage. Additionally, we conduct a comparative analysis of the number of unit test candidates. We use DeepSeek-Coder and RATester (with DeepSeek-Coder as the backbone model) to generate test cases for all projects, setting the number of unit test candidates from 1 to 10 (i.e., generating 1 to 10 test cases for each focal method or function). We then calculate the average line coverage across all projects.

Results. We discuss the results from three perspectives: the total generation time across all projects, the context retrieval efficiency and relevance analysis, and the impact of the unit test candidate number.

Total generation time across all projects. Fig. 5 shows the total generation time required by each baseline and RATester for unit test generation. According to the results, we observe that: (1) RATester requires more generation time compared to the basic LLMs. For example, CodeLlama alone takes 18.2 hours, while RATester built on CodeLlama requires 25.6 hours. Overall, the basic LLMs need between 18.2 and 19.9 hours, whereas RATester requires between 25.6 and 28.7 hours. However, we believe this additional time is acceptable in practical usage due to the higher compile rates, increased line coverage, and a greater number of killed mutants achieved by RATester (refer to Section V-B for more details). (2) The three basic LLMs (i.e., CodeLlama, DeepSeek-Coder, and Magicoder) perform fast, while the others perform relatively a little slow. More precisely, CodeLlama, DeepSeek-Coder, and Magicoder only take 18.2 hours, 19.1 hours, and 19.9 hours to generate unit tests for all projects. Among all the approaches, ChatUniTest has the longest generation time, taking 45.9 hours. This is because ChatUniTest includes redundant context and also features a unit test repair process that requires multiple iterations to arrive at the final unit test.

Context retrieval efficiency and relevance analysis. We conduct a comprehensive analysis of RATester's context usage efficiency. We measure the token length of all contexts retrieved by RATester during unit test generation and calculate the average context size per focal method and function. Our results show that RATester requires an average of 749 tokens as context for generating unit tests for each focal method and function, representing a 31.28% reduction compared to ChatUniTest (1,090 tokens for context). This reduction is achieved because RATester only introduces context that is actively used during

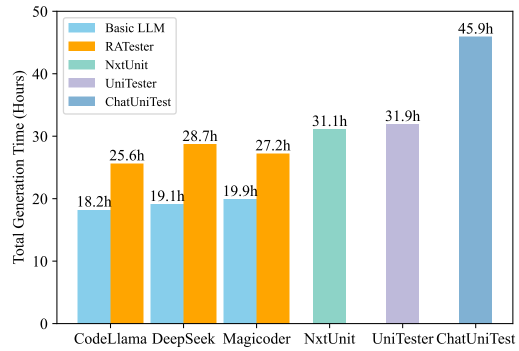

Fig. 5: RQ-3: Total generation time of RATester and baselines

unit test generation, unlike ChatUniTest, which retrieves dependency information from the focal method/function and class that may produce redundant and unused context.

We also measure the relevance of retrieved context by analyzing whether Gopls successfully retrieves definitions and whether RATester ultimately utilizes these definitions (hit rate). Statistical results demonstrate that Gopls successfully retrieves definitions in 89% of queries, and RATester's 86% hit rate for retrieved definitions indicates that a significant portion of these definitions are actively utilized in subsequent test generation.

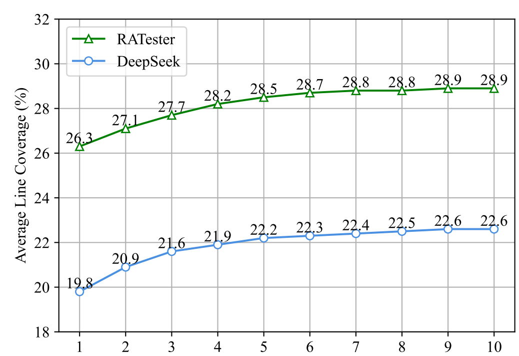

Fig. 6: RQ-3: Performance of RATester and DeepSeek-Coder across varying unit test candidate number

Impact of the unit test candidate number. According to the results on the Fig. 6, we find that: (1) Different candidate numbers have varying impacts on the performance of RATester and DeepSeek-Coder, with both models showing improved performance as the number of candidates increases. (2) The curve indicates that the performance of RATester far exceeds that of DeepSeek-Coder. RATester only requires generating one unit test for each focal method or function to achieve higher line coverage than DeepSeek-Coder, which requires ten unit tests. This demonstrates that RATester is more efficient and produces higher-quality unit tests with a less number of candidates. (3) Increasing the number of candidates does not guarantee significant performance improvements. As we continuously increase the number of candidates from 2 to 10, the performance of both RATester and DeepSeek-Coder improves only slightly, while the generation cost with the LLM increases significantly. Considering both the performance improvements and the associated generation costs, we adopt a candidate number of 1 unit test as the default setting.

Answer to RQ-3: *(1) RATester requires more generation time than the basic LLMs; however, considering the performance improvements, this additional time is justified. (2) Increasing the candidate number can enhance the performance of RATester, but the improvement is not significant.*

D. RQ-4: Generalizability of RATester

Objective. In the previous RQs, we demonstrate the effectiveness and efficiency of RATester on Golang projects using DeepSeek-Coder and other LLMs as backend models. Besides the Go language, RATester generalizes to other programming languages that offer language server support for retrieving contextual information (e.g., jdtls [19] for Java and pyls [20] for Python). To further validate the broad applicability of our approach, this RQ investigates the generalizability of RATester across two dimensions: (1) cross-language compatibility with Java and Python and (2) scalability to larger LLMs with varying parameter sizes. Our objective is to demonstrate that RATester's language server-based context injection approach generalizes effectively beyond Golang and maintains superior performance across different model scales.

Experimental Design. We conduct two experiments to evaluate RATester's generalizability. First, we extend RATester to support Java and Python by integrating jdtls [19] and pyls [20] as their respective language servers. For the Java language, we adopt the same four projects at the same versions used in ChatUniTest [6]: Cli, Csv, Ecommerce, and Binance. For the Python language, we use the fixed versions corresponding to the first bug of 17 projects from the BugsInPy dataset [21], ensuring that no tests failed within the projects. As these projects contain a large number of methods and functions, we randomly sample a subset from each project to balance evaluation cost with statistical significance. The sample size is determined based on a confidence level of 95% and a confidence interval of 5%. In total, we extract 1,179 methods for the Java language and 3,883 methods and functions for the Python language.

Second, we evaluate RATester's scalability across different LLM sizes by evaluating four variants: RATester_C1 uses CodeLlama 7B as the backend model, RATester_C2 uses CodeLlama 34B, RATester_D1 uses DeepSeek-Coder 6.7B, and RATester_D2 uses DeepSeek-Coder 33B.

Results. We analyze the results from two perspectives: cross-language compatibility and scaling with larger LLMs.

Cross-language compatibility. Table VII presents the results across Java and Python projects. RATester consistently achieves the highest performance across both languages. Specifically, in Java projects, RATester achieves 33.25% to 75.14% line coverage, outperforming baselines that range from 8.69% to 72.90%. On the BugsInPy (Python), RATester demonstrates improvements of 5.45% to 284.21% compared to the baselines, achieving 59.36% average line coverage. These results indicate that RATester's language server-based approach generalizes effectively beyond Golang to other programming languages.

TABLE VII: RQ-4: RATester *vs.* Baselines across different programming languages in line coverage

Projects	UniTester	ChatUniTest	ChatTester	SymPrompt	RATester
Cli	21.27%	66.39%	60.70%	69.45%	72.86%
Csv	25.65%	70.37%	63.28%	72.90%	75.14%
Ecommerce	8.69%	26.11%	25.88%	28.69%	33.25%
Binance	12.69%	47.92%	48.36%	50.92%	52.08%
BugsInPy	15.45%	53.17%	51.10%	56.29%	59.36%

Scaling with larger LLMs. Table VIII presents the performance comparison across different LLM scales. Our experiments demonstrate that deploying RATester on larger LLMs consistently yields improved performance across all three programming languages. Upgrading from CodeLlama 7B to 34B results in performance improvements of 20.83% in Golang, 16.57% in Java, and 4.39% in Python. Similarly, scaling from DeepSeek-Coder 6.7B to 33B achieves improvements of 22.06% in Golang, 16.10% in Java, and 5.85% in Python. However, this performance gain comes with significant increases in computational overhead. Considering the trade-off between performance and resource requirements, 6.7B to 7B LLMs provide a practical balance for widespread adoption while maintaining competitive performance.

TABLE VIII: RQ-4: RATester with different LLM sizes across different programming languages in line coverage

Models	Golang	Java	Python
RATester_C1	23.00%	51.13%	51.96%
RATester_C2	27.79%	59.60%	54.24%
RATester_D1	26.25%	58.33%	59.36%
RATester_D2	32.04%	67.72%	62.83%

Answer to RQ-4: *RATester demonstrates strong generalizability across multiple dimensions. The approach successfully extends to Java and Python with consistent performance improvements and scales effectively to larger LLMs with proportional performance gains.*

VI. Discussion

A. Evaluation of Potential Data Leakage

1) Analysis with data after the training cutoff date: To mitigate potential data leakage, we select DeepSeek-Coder as the backend model for RATester, noting that the pre-training data for DeepSeek was collected from GitHub before February 2023 [12]. We evaluate potential data leakage using two post-cutoff datasets: (1) the Adk project (Java), a Google-designed project created after January 1, 2024, with over 500 stars, from which we extract 773 focal methods; and (2) 500 randomly selected focal methods/functions from Golang projects with commits containing updates, modifications, or creation dates after the model's training cutoff. We conduct comparative evaluations between RATester and the baseline DeepSeek-Coder on these datasets to mitigate potential data leakage concerns. The results demonstrate that RATester achieves 54% and 32% line coverage on Java and Golang, respectively, significantly outperforming DeepSeek-Coder's 38% and 19% coverage. These findings indicate that RATester's superior performance stems from its methodological improvements rather than data leakage, validating the robustness of our approach.

2) Analysis of similarity and contribution: We calculate the number of unit tests generated by RATester, which matches the reference unit test in all eight Golang projects. We find that out of 11,195 generated unit tests, only 1 of these aligns with the unit tests in projects. Additionally, compared to the basic LLMs (i.e., CodeLlama, DeepSeek-Coder, and Magicoder), RATester demonstrates a significant enhancement in performance, achieving an increase in line coverage of 22.3% to 34.1%. Furthermore, the unit tests generated by RATester also contribute to completing the original unit tests in the projects, raising the line coverage from 56.49% to 60.98%. This demonstrates that the improved results achieved by RATester are not merely a result of memorizing the training data.

B. Qualitative Assessment of RATester

To complement our quantitative evaluation, we conduct a comprehensive qualitative analysis of RATester across two dimensions: test quality assessment and failure pattern analysis. We randomly sample 200 focal methods/functions from Golang projects, along with all corresponding unit tests generated by RATester, for detailed assessment.

1) Test Quality Assessment: We conduct an expert evaluation involving 5 experienced Golang developers with extensive development backgrounds. For readability and maintainability, we establish evaluation scores ranging from 1-5, where 1 represents the worst and 5 represents the best. The evaluation results demonstrate that RATester-generated unit tests achieve average scores of 4.28 for readability and 4.10 for maintainability, indicating high-quality test generation.

2) Failure Pattern Analysis: We systematically analyze failure cases where RATester produces incorrect tests. Our analysis reveals that these failures primarily stem from runtime execution issues: panic errors and timeout collectively account for 54% of all failure cases. The remaining failures include context extraction failures, LLM context window constraints, and cases where excessively long context prevents the model from focusing on critical information, resulting in compilation errors. These findings inform future enhancement strategies for improving robustness in edge cases and optimizing context management.

C. Comparison with Retrieval-Augmented Generation (RAG)

Similar to LLM-based baselines, RAG methods fail by retrieving context based solely on the method and class under test, leading to redundancy or omissions of critical context information. To validate the superiority of RATester, we conduct comparative evaluations against two representative RAG methods: embedding-based retrieval using CodeBERT [22] and BM25 [23]. We ensure fair comparison by maintaining equivalent context quantities across all methods and evaluating performance on the same datasets. Table IX presents the comparative results across three programming languages in terms of line coverage. Our evaluation demonstrates that RATester consistently outperforms both RAG methods across all tested languages. Specifically, RATester achieves performance improvements of 16.61% and 22.78% over embedding-based and BM25 methods respectively in Golang, 15.28% and 24.66% in Java, and 14.07% and 13.17% in Python. These results validate that precise, on-demand context injection through language servers significantly outperforms RAG methods.

TABLE IX: RATester *vs.* RAG across different programming languages in line coverage

Languages	Embed.	BM25	RATester
Golang	22.51%	21.38%	26.25%
Java	50.60%	46.79%	58.33%
Python	52.04%	52.45%	59.36%

D. Threats to Validity

Internal Validity. The concern relates to the efficiency of RATester in practical deployment scenarios. The efficiency concerns stem from the language server's retrieval time and the computational cost of processing extensive context information. To address these limitations, future work could implement performance optimization strategies such as context caching, incremental retrieval, and intelligent context pruning. These optimizations would enhance the practical applicability of RATester while maintaining its effectiveness in generating high-quality unit tests.

External Validity. The main external threat to validity relates to scalability challenges when applying RATester to large-scale repositories. As repository size increases, the language server may experience longer retrieval times for context information, and the retrieved context may become excessively large, potentially overwhelming the LLM's context window. Additionally, large repositories often contain complex interdependencies between modules, which may require more sophisticated strategies for relevant context selection and prioritization. While our current evaluation focuses on moderately-sized projects, future work should investigate the scalability limitations and develop optimization strategies for handling large-scale codebases, including techniques such as hierarchical retrieval and distributed processing approaches.

VII. Related Work

A. Unit Test Generation

Unit test generation approaches can be classified into three main categories: traditional, learning-based, and LLM-based approaches.

Traditional approaches [24], [25] primarily focus on maximizing code coverage through tools like Randoop [25] and EvoSuite [24]. While effective at achieving high coverage [26], [27], these approaches fail to produce well-written, maintainable unit tests for practical developer use [9], [28].

Learning-based approaches [10], [28]–[31] address traditional limitations by leveraging neural models trained on large-scale code datasets. Representative works include AthenaTest [28] and A3Test [29], which fine-tune pre-trained language models on the Methods2Test dataset, TOGA [30] for test oracle inference, and UniTester [10] for multi-language test synthesis. However, these approaches remain heavily dependent on task-specific datasets extracted from open-source repositories.

In response to the challenges posed by learning-based approaches, researchers are increasingly utilizing LLMs to generate unit tests directly from contextual information, reducing reliance on task-specific datasets. Recent works include CodaMOSA [32] for overcoming coverage plateaus and ChatGPT-based approaches [4]. However, LLMs exhibit hallucinations due to limited project-specific knowledge, manifesting as non-existent method calls and incorrect parameter assignments. To address this, researchers have explored context extraction techniques: ChatTester [5] employs iterative generation with execution feedback and extracted context, while ChatUniTest [6] introduces adaptive focal context mechanisms and generation-validation-repair processes. Recent studies [7], [8] further investigate focal and dependency context roles in improving LLM-based test generation. These approaches can lead to two major problems. First, they may not capture all the context needed for unit test generation, resulting in missing context if the required information cannot be found within the focal method and class. Second, they may extract redundant context by retrieving dependency information that is not ultimately used during the unit test generation process.

Different from existing works, RATester employs a more effective, dynamic strategy. It searches for the necessary context using identifiers that are already generated within the unit test snippet. This approach ensures that the context required for unit test generation is not lost and is highly relevant.

B. Pre-trained Language Model

Pre-trained language models have gained widespread adoption due to their training on vast datasets with billions of parameters, leading to remarkable performance improvements across diverse applications. These models are highly adaptable to downstream tasks through fine-tuning [33], [34] and prompting [4], [35]–[37], with versatility arising from extensive pre-training that provides a robust knowledge base. Fine-tuning adjusts model parameters for specific tasks through iterative training on dedicated datasets, while prompting offers a more direct approach by providing task-specific instructions in natural language without parameter adjustments.

These models are typically based on the transformer architecture [38] and categorized into three types: encoder-only, encoder-decoder, and decoder-only. Encoder-only models (e.g., CodeBERT [22], GraphCodeBERT [39]) and encoder-decoder models (e.g., PLBART [40], CodeT5 [41]) use objectives like Masked Language Modeling (MLM) or Masked Span Prediction (MSP), where masked tokens are predicted from context. Trained on diverse code-related data, these models are then fine-tuned for specific tasks to achieve enhanced performance [42]–[44]. Decoder-only models have gained significant attention through causal language modeling objectives, predicting next tokens based on previous context. GPT [33] and its variants exemplify this architecture, marking a pivotal point in widespread LLM applications.

To enhance LLMs' generalization and alignment with human intentions on previously unseen downstream tasks, recent research has focused on instruction tuning and reinforcement learning to improve model performance [45]–[47]. For instance, OpenAI's ChatGPT [48] exemplifies this approach, combining instruction tuning with reinforcement learning from human feedback. Open-source instructed LLMs, such as CodeLlama [11], DeepSeek-Coder [12], and Magicoder [13], also demonstrate promising performance and broader adaptability [4], [34], [49].

VIII. Conclusion

This paper addresses the challenge of LLM hallucinations in unit test generation by proposing RATester, which enhances LLMs' ability to generate repository-aware unit tests through precise context injection. To provide LLMs with global knowledge comparable to that of human developers, RATester integrates language servers to enable dynamic definition lookup capabilities. When encountering unfamiliar identifiers during test generation, RATester proactively leverages language servers to fetch relevant definitions and documentation comments, thereby preventing erroneous usage and reducing hallucinations. We evaluate RATester's effectiveness and efficiency by constructing a comprehensive Golang dataset from real-world projects and conducting comparative analysis against baselines. The results illustrate the advantages of RATester over these baselines. Additionally, our model-agnostic and generalizability analysis confirms RATester's effectiveness across different models, programming languages, and model scales, validating its broad applicability.

Acknowledgements

This work was supported by Zhejiang Pioneer (Jianbing) Project (2025C01198(SD2)), the National Natural Science Foundation of China (No.62572429, 62202419), the Fundamental Research Funds for the Central Universities (No.226-2022-00064), Zhejiang Provincial Natural Science Foundation of China (No.LY24F020008), the Key Research and Development Program of Zhejiang Province (No.2021C01105), the State Street Zhejiang University Technology Center, and Douyin Co., Ltd.

References

[1] V. Garousi and J. Zhi, "A survey of software testing practices in canada," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1354–1376, 2013.

[2] J. Lee, S. Kang, and D. Lee, "Survey on software testing practices," *IET software*, vol. 6, no. 3, pp. 275–282, 2012.

[3] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[4] C. Ni, X. Wang, L. Chen, D. Zhao, Z. Cai, S. Wang, and X. Yang, "Casmodatest: A cascaded and model-agnostic self-directed framework for unit test generation," *arXiv preprint arXiv:2406.15743*, 2024.

[5] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1703–1726, 2024.

[6] 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*, 2024, pp. 572–576.

[7] 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*, vol. 1, no. FSE, pp. 951–971, 2024.

[8] S. Gao, C. Wang, C. Gao, X. Jiao, C. Y. Chong, S. Gao, and M. Lyu, "The prompt alchemist: Automated llm-tailored prompt optimization for test case generation," *arXiv preprint arXiv:2501.01329*, 2025.

[9] S. Wang, X. Mao, Z. Cao, Y. Gao, Q. Shen, and C. Peng, "Nxtunit: Automated unit test generation for go," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, 2023, pp. 176–179.

[10] Y. He, J. Huang, Y. Rong, Y. Guo, E. Wang, and H. Chen, "Unitsyn: A large-scale dataset capable of enhancing the prowess of large language models for program testing," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1061–1072.

[11] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[12] D. AI, "Deepseek coder: Let the code write itself," https://github.com/deepseek-ai/DeepSeek-Coder, 2023.

[13] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: empowering code generation with oss-instruct," in *Proceedings of the 41st International Conference on Machine Learning*, 2024, pp. 52 632–52 657.

[14] "Replication," 2025. [Online]. Available: https://github.com/vinci-grape/RATester

[15] "gopls," 2025. [Online]. Available: https://github.com/golang/tools/tree/master/gopls

[16] "gremlins," 2025. [Online]. Available: https://github.com/go-gremlins/gremlins

[17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[18] "Hugging face," 2025. [Online]. Available: https://huggingface.co

[19] "jdtls," 2025. [Online]. Available: https://github.com/eclipse-jdtls/eclipse.jdt.ls

[20] "pyls," 2025. [Online]. Available: https://github.com/python-lsp/python-lsp-server

[21] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh *et al.*, "Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1556–1560.

[22] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, pp. 1536–1547, 2020.

[23] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[24] 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*, 2011, pp. 416–419.

[25] 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*, 2007, pp. 815–816.

[26] A. Aleti, I. Moser, and L. Grunske, "Analysing the fitness landscape of search-based software testing problems," *Automated Software Engineering*, vol. 24, pp. 603–621, 2017.

[27] C. Oliveira, A. Aleti, L. Grunske, and K. Smith-Miles, "Mapping the effectiveness of automated test suite generation techniques," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 771–785, 2018.

[28] 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.

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

[30] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: A neural method for test oracle generation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2130–2141.

[31] L. Saes, "Unit test generation using machine learning," *Universiteit van Amsterdamg*, 2018.

[32] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 919–931.

[33] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[34] X. Yin, C. Ni, and S. Wang, "Multitask-based evaluation of open-source llm on software vulnerability," *IEEE Transactions on Software Engineering*, 2024.

[35] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[36] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "Thinkrepair: Self-directed automated program repair," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1274–1286.

[37] X. Yin, C. Ni, T. N. Nguyen, S. Wang, and X. Yang, "Rectifier: Code translation with corrector via llms," *arXiv preprint arXiv:2407.07472*, 2024.

[38] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[39] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations*.

[40] W. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "Unified pre-training for program understanding and generation." in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021.

[41] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *EMNLP 2021-2021 Conference on Empirical Methods in Natural Language Processing, Proceedings*. Association for Computational Linguistics (ACL), 2021, pp. 8696–8708.

[42] C. Ni, X. Yin, K. Yang, D. Zhao, Z. Xing, and X. Xia, "Distinguishing look-alike innocent and vulnerable code by subtle semantic representation learning and explanation," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1611–1622.

[43] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.

[44] D. Hin, A. Kan, H. Chen, and M. A. Babar, "Linevd: Statement-level vulnerability detection using graph neural networks," in *Proceedings of the 19th international conference on mining software repositories*, 2022, pp. 596–607.

[45] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," *Advances in neural information processing systems*, vol. 30, 2017.

[46] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[47] D. M. Ziegler, N. Stiennon, J. Wu, T. B. Brown, A. Radford, D. Amodei, P. Christiano, and G. Irving, "Fine-tuning language models from human preferences," *arXiv preprint arXiv:1909.08593*, 2019.

[48] OpenAI, "Chatgpt: Optimizing language models for dialogue. (2022)," https://openai.com/blog/chatgpt/, 2022.

[49] X. Yin, C. Ni, X. Xu, and X. Yang, "What you see is what you get: Attention-based self-guided automatic unit test generation," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2025, pp. 1039–1051.