LSPAI: An IDE Plugin for LLM-Powered Multi-Language Unit Test Generation with Language Server Protocol

Gwihwan Go
BNRist, Tsinghua University
Beijing, China
iejw1914@gmail.com

Chijin Zhou*
BNRist, Tsinghua University
Beijing, China
tlock.chijin@gmail.com

Quan Zhang
BNRist, Tsinghua University
Beijing, China
quanzh98@gmail.com

Yu Jiang*
BNRist, Tsinghua University
Beijing, China
jiangyu198964@126.com

Zhao Wei
Tencent
Beijing, China
zachwei@tencent.com

Abstract

Unit testing is crucial for ensuring code validity, and extensive research has been conducted to advance this domain. However, existing studies fail to address critical industry requirements, particularly support for multi-language static analysis and real-time unit test generation. While integrating static analysis with a Large Language Model could address these challenges, it typically requires significant manual effort to implement across diverse programming languages. To address this, we propose LspAi, an automated unit test generation tool that leverages well-established language analysis tools and integrates them into a unified development environment via the Language Server Protocol. This approach equips LLM with multi-language static analysis capabilities, allowing a single tool to support systematic unit test generation across multiple languages. We evaluate our method by comparing line coverage across different LLMs and programming languages, demonstrating both superior performance and broad applicability. On real-world projects, LspAi achieved line coverage improvements of 145% for Java, 931% for Golang, and 95.62% for Python compared to Copilot. In addition, we also share our lessons learned from applying the tool in Tencent Ltd.

ACM Reference Format:

Gwihwan Go, Chijin Zhou, Quan Zhang, Yu Jiang, and Zhao Wei. 2025. LSPAI: An IDE Plugin for LLM-Powered Multi-Language Unit Test Generation with Language Server Protocol. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3696630.3728540

1 Introduction

Unit testing plays a pivotal role in modern software development by ensuring the validity and reliability of code. As software systems grow in complexity, the importance of robust unit tests cannot be overstated, serving as a fundamental practice for identifying defects early and facilitating maintainable codebases. Extensive research has been dedicated to automating unit test generation, leading to the development of Search-Based Software Testing (SBST) tools such as EvoSuite [12], Randoop [30], and Pynguin [25]. More recently, the evolution of Large Language Models (LLMs) has introduced a new paradigm for unit test generation. Models like GPT [7] and Copilot [14] can understand the context of the code and generate semantically relevant unit tests, significantly improving software development efficiency.

Despite significant advancements, LLMs are still prone to generating incorrect unit tests. For example, Copilot, one of the most popular tools used by many companies, is still capable of making mistakes, as acknowledged by the Copilot development team [26]. Similarly, Siddiq et al. [33] found that LLM-generated test cases yield a relatively low validity rate, ranging from 2% to 12.7%, based on the SF110 [13] benchmark. As a result, researchers have proposed integrating static analysis with LLMs to help them better understand context and generate more accurate unit tests [21, 39, 41]. However, current research does not address the following two fundamental requirements of the software industry, which limits the broader adoption of these approaches in real-world settings.

First, performing static analysis across multiple programming languages is challenging. Industries adopt a variety of programming languages for different projects, and a test case generator should ideally support multiple languages. However, as shown in Table 1, most academic research has focused on one specific language rather than multi-language support. This focus stems from the difficulty of performing unified static analysis across diverse languages. Therefore, developers are forced to build customized analysis pipelines for each language, which requires significant

Table 1: Literature analysis which shows current research gap on unit test generation.

Tools	Real-Time	Multi-Language	Static Analysis Support			
			Java	Python	Golang	Others
UTGen [6], EvoSuite [12, 23, 43], Randoop [30], HITS [39] , casmoda [29], testspark [32], ChatUniTest [41]	✗	✗	✓	✗	✗	✗
PynGuin [25], CodaMosa [21], CoverUp [3], MuTAP [5] , TELPA [42], SymPrompt [31], CLAP [37]	✗	✗	✗	✓	✗	✗
NxtUnitGo [38]	✗	✗	✗	✗	✓	✗
Copilot [14]	✓	✓	✗	✗	✗	✗
LspAi	✓	✓	✓	✓	✓	✓

*Yu Jiang and Chijin Zhou are corresponding authors.

FSE Companion '25, June 23–28, 2025, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728540

manual adaptation effort. As a result, as far as we know, there are currently no academic tools available that can generate multi-language unit tests using static analysis.

Second, it is challenging to support real-time unit test generation when integrating static analysis. Developers often write unit tests concurrently with writing code. However, current SBST tools and LLM-integrated tools are unsuitable for scenarios requiring instant test generation, as they typically require the compilation of entire projects to perform static analysis and collect coverage feedback. Consequently, the reliance of SBST tools on coverage feedback to enhance test case quality limits their feasibility for real-time use. This issue also persists in recent LLM-integrated tools [3, 4, 21, 22, 39, 41], which depend on heavy static analysis and coverage feedback to mitigate LLM hallucinations. As a result, no academic tool currently supports real-time unit test generation, as illustrated in Table 1.

To tackle the aforementioned challenges, we introduce LspAi, a real-time unit test generation tool powered by LLMs and integrated with static analysis for multi-language codebases. Our key insight is that well-established language analysis tools exist for each programming language and can be accessed through the Language Server Protocol (LSP) in a unified way. By leveraging the LSP, we can perform lightweight static analysis on multiple languages within a single environment with minimal effort. Specifically, LspAi operates in two main steps: First, LspAi conducts dependency analysis by extracting key tokens from the focal method and retrieves the corresponding dependent source code. Second, using the retrieved dependency source code, LspAi performs real-time unit test generation and fixing. This approach effectively leverages reliable static analysis tools to improve LLMs' ability.

LspAi brings two main benefits to developers who work in industries. First, LspAi supports real-time unit test generation *without whole project compilation*, allowing developers to generate unit tests concurrently with code writing. Second, LspAi simplifies the setup process by only requiring *a simple installation* of relevant language analysis plugins (*e.g.,* extensions for Visual Studio Code), making it easily adaptable to various programming languages.

We developed LspAi as an IDE (Integrated Development Environment) plugin for seamless integration and evaluated its performance across three widely used programming languages: Java, Python, and Golang. Our evaluation shows that LspAi consistently improves unit test performance in terms of line coverage across real-world projects, regardless of programming language. Compared to Copilot, LspAi achieved line coverage improvements of 145% for Java, 931% for Golang, and 95.62% for Python. When compared to a naive LLM implementation, the improvements were 122% for Java, 2,003% for Golang, and 4.84% for Python. Additionally, we share practical insights and lessons learned from applying LspAi in an industrial setting at Tencent Ltd.

- We identified a research gap in current unit test generation: the lack of support for multi-language codebases and real-time test generation scenarios.
- We designed LspAi as an IDE plugin, a practical tool that generates effective unit tests across multiple programming languages. The source code can be found at https://github.com/Gwihwan-Go/LSPAI/tree/fse-industry.

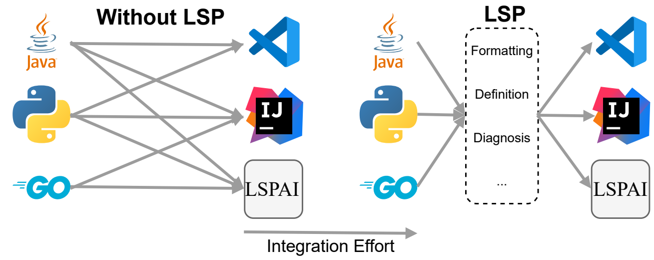

Figure 1: Integration effort before and after LSP.

- We evaluated LspAi in real-world projects written in three programming languages, demonstrating its ability to consistently improve unit test performance. LspAi LspAi achieved line coverage improvements of 145% for Java, 931% for Golang, and 95.62% for Python compared to Copilot.

2 Language Server Protocol

Language Server Protocol. The Language Server Protocol (LSP) was introduced by Microsoft in 2016 [27] to solve a growing pain point: every editor and IDE needed its own, hand-rolled plug-in for each programming language. As the number of editors and languages exploded, this approach became unsustainable—language authors had to rewrite the same analysis logic over and over, while tool vendors struggled to keep pace with new languages. LSP emerged as a common, JSON-RPC-based "bridge" that lets any editor talk to an external language server that already knows how to parse and analyze code.

Before vs. After LSP. As illustrated in Figure 1, two processes exchange JSON-RPC messages in the LSP usage scenario: the *language client*—the editor or IDE a developer is using, shown on the right in each sub-figure—and the *language server*, a standalone process that performs parsing, static analysis, and other language-specific tasks, shown on the left. Before LSP, adding rich language support meant duplicating effort: Eclipse's Java plug-in, VS Code's JavaScript plug-in, Vim's Python plug-in, and so on—each a separate codebase re-implementing the same features. After LSP, the split is clean: editors forward user actions (e.g., `didOpen`, `didChange`) to a single language server, which then returns results (e.g., completions, diagnostics). One server can now power many editors, and one editor can support many languages simply by connecting to different servers.

Typical LSP capabilities. A standard LSP server typically offers *code completion*, *go-to definition*, *find references*, *real-time diagnostics* (errors and warnings as you type), *hover documentation*, *rename symbol*, *code actions* (quick fixes and refactors), *formatting*, and, in newer versions, *semantic highlighting*. Because these capabilities travel over the same protocol, users obtain a consistent, IDE-grade experience in any LSP-aware editor with minimal setup.

3 Design of LspAi

This section describes the design of LspAi, a unit test generation tool that enhances unit test creation through multi-language static analysis aided by LSP. Figure 2 illustrates LspAi's overall workflow. When the developer requests unit test generation for a specific

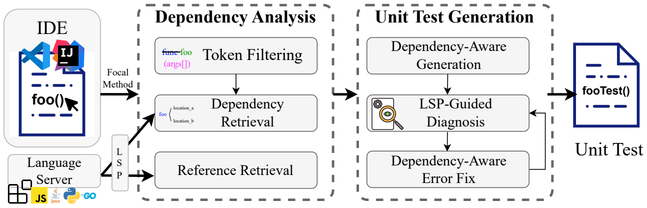

Figure 2: Overall workflow of LspAi.

method, LspAi generates a unit test following two steps. First, LspAi collects dependency information for the given method by extracting and retrieving token definitions. Second, it generates the unit test based on the collected dependency information. The generated unit test is then analyzed using LSP. If any issues are detected, LspAi retrieves the necessary dependencies and corrects the errors.

3.1 Employed LSP Features

LspAi leverages the advantages of LSP, conducting static analysis using features provided by the language server. In this way, LspAi can do consistent analysis across different IDEs and programming languages with the unified pipeline. Specifically, LspAi utilizes the following features provided by the language server.

Symbol Provider. In LSP, symbols represent code entities such as files, modules, classes, functions, and variables within the source code. When LspAi requests symbols for a specific file path, the language server returns a hierarchical structure of these symbols found in the given text document. LspAi leverages the *Symbol Provider* to identify unit test entries through these symbols and to locate variable definitions by traversing the collected symbols.

Semantic Token Provider. Tokens are the smallest elements of code that can be broken down [28]. Semantic tokens extend tokens by adding contextual information, utilizing language servers that deeply understand the source file. When LspAi requests semantic tokens for a specific range, the language server returns an array of objects, each containing the context information of the token and its range. The *Semantic Token Provider* allows LspAi to determine how each token is used, facilitating a more granular and precise analysis of methods, variables, and other constructs.

Definition Provider. This feature plays a crucial role in analyzing dependencies for a target function by identifying the locations of function or class declarations. When LspAi requests the *Definition Provider* for a specific token, it returns the locations of token definitions. This capability allows LspAi to accurately map dependencies within the codebase, ensuring comprehensive analysis of the target functions or classes.

Reference Provider. The *Reference Provider* enables LspAi to identify all occurrences of a particular symbol within the codebase. By requesting references for a specific symbol, the language server returns a list of locations where the symbol is used. This feature is essential for understanding the context and usage patterns of the symbol, which aids in accurately determining dependencies and ensuring that generated unit tests cover relevant interactions.

Diagnosis Provider. It is vital for increasing the validity of the generated code. When LspAi requests a diagnosis, the *Diagnosis Provider* provided by the language server analyzes the code using its understanding of the source, detects any warnings or errors, and provides their specific locations. This allows LspAi to effectively identify and rectify issues in the generated code.

3.2 Dependency Analysis

This module gathers dependency information to generate reliable unit tests with high coverage. The dependency information is collected in three steps: token filtering, dependency retrieval, and reference retrieval. Through this process, LspAi acquires streamlined, high-quality dependency information.

Token Filtering. This step enhances the quality of the data to be collected by extracting tokens that are more likely to be relevant to the focal method. A focal method that requires testing is typically complex, containing numerous tokens. Analyzing every token of the method would generate a large amount of unnecessary data, most of which would not contribute to unit test generation. For example, the `parse` method in the `Parser` class of the commons-cli [8] project contains over 100 tokens within approximately 40 lines of code. Analyzing and retrieving information for over 100 tokens is inefficient and does not effectively enhance the quality of unit tests. Therefore, appropriate token filtering is essential. The token filtering strategy of LspAi involves two main steps: *(1) Selecting Key Tokens*: LspAi consider a token is important if it is given by the argument value of the method or is returned by the method. *(2) Selecting Associated Tokens*: LspAi determine whether the tokens are associated with key tokens, utilizing the knowledge of the language server. Specifically, it requests the role of tokens that co-located with the key tokens by examining their types and modifiers through *Semantic Token Provider*. A co-located token is considered meaningful if the language server considers the token's role as declaring or defining. Following the above steps, the 100 tokens under `parse` method can be streamlined to 10 tokens. Ultimately, this module returns the extracted tokens, which LspAi uses to retrieve further information.

Dependency Retrieval. This step involves a strategy to extract relevant dependency information from the given tokens, ensuring LspAi retains only essential data for unit test generation while discarding unnecessary details from the language server. This is important because retrieved dependency information is often verbose, including comments, unrelated properties, or large code snippets. This can hinder unit test generation and degrade LspAi's performance. To address this, we apply heuristic rules based on LSP knowledge. First, by requesting the *Definition Provider* using the token's position, we collect the symbol that defines the token. Next, using the *Symbol Provider*, we identify the symbol's type (*e.g.*, function, class, method, variable, or property). Finally, we summarize the relevant code snippet based on the symbol type. For example, functions are summarized by their return type and input arguments, while methods are summarized with their return type, input arguments, and associated class member properties.

Reference Retrieval. Refering to the use case of the focal method can enhance the correctness of generated test codes. Especially for LLM, which determines its output based on context, much research [15, 40] has proved that giving an example can enhance the quality of its output. In this regard, LspAi collects every use case of the focal method, utilizing *Reference Provider*. The collected reference information is then passed to the next step along with the dependency information and is used to generate unit test code.

3.3 Unit Test Generation

This module is responsible for generating reliable unit tests with high coverage without compiling or executing code. It leverages the given dependency information and LLM to generate unit tests. Subsequently, to mitigate the limitations of LLM, it detects issues in the generated code and fixes them.

Dependency-Aware Generation. LspAi generates unit tests by incorporating information passed by the section 3.2. In detail, We construct the prompt incorporating the source code of the focal method, natural language description, and retrieved information.[1] We construct our prompt template based on that of ChatUniTest [41]. Since this template is Java-specific, for generating unit tests in other programming languages, we slightly modify the prompt accordingly. The final prompt ranges from 1000 to 1,500 tokens, depending on the length of the focal method. The constructed prompt is then provided to a LLM to produce the unit test.

LSP-Guided Diagnosis. This component detects issues in the generated test code in real-time without the need for compilation or execution. LLMs can produce syntactically incorrect or incompilable code due to hallucinations. Hallucination [17, 18] refers to the generation of syntactically incorrect or semantically invalid code that deviates from the desired output. However, in a real-time generation setting, where compilation or execution is not feasible, we need alternative methods to mitigate the LLM's hallucinations. LspAi utilizes *Diagnosis Provider* supported by LSP to inspect the generated code. If *Diagnosis Provider* does not detect any issues, LspAi saves the unit test code, if there is any issue, LspAi collects them and prioritize based on severity.

Dependency-Aware Error Fix. This step integrates dependency information to effectively fix errors. Based on the diagnosis, LspAi identifies the related symbol and retrieves the necessary dependency information using the *Symbol Provider*. This information is then incorporated into the prompt to assist the LLM in correcting the error alongside the necessary dependency source code. The constructed prompt is sent to the LLM to fix the code. After the fix is made, LspAi returns to the *LSP-Guided Diagnosis* step to verify whether the issue has been resolved. If the error is fixed or the iteration limit is reached, the corrected code is saved and presented to the developers.

4 Evaluation

In this section, we comprehensively evaluate LspAi's performance on real-world projects across different programming languages.

4.1 Experiment Setup

Programming languages. We selected three different programming languages, Python, Java, and Golang, for the experiment. We selected Python and Java because their unit test generation capabilities have been extensively studied in previous research. Golang was chosen for two reasons: (1) it is widely used in industry but has received relatively little attention in academic studies, and (2) as a relatively new language, we anticipate it presents unique challenges for LLMs in generating valid code.

Table 2: Dataset Statistics

Project	Abbr.	Domain	Version	Language
Commons-CLI [8]	CLI	Cmd-line Interface	eb541428	Java
Commons-CSV [9]	CSV	Csv file Processing	92e486ac	Java
Logrus [34]	LOG	logging for Golang	d1e6332	Golang
Cobra [11]	COB	Golang CLI interactions	3a6873e	Golang
Black [10]	BAK	Python code formatter	8dc9127	Python
Crawl4AI [36]	C4AI	LLM Friendly Web Crawler	8878b3d	Python

Baseline Selection. We selected the baselines that satisfy below conditions:(1) they support unit test generation for multiple programming languages, and (2) they generate unit test code directly without compiling the entire project. Most of the tools listed in Table Table 1 do not meet these criteria, except for Copilot. However, the current version of Copilot only supports a limited set of LLMs [2] (specifically, GPT-4o), which limits us to compare different LLMs. To address this, we built a baseline using the same prompt template as LspAi, but without including dependency information from LSP. We call this version Naive. Ultimately, we evaluated the performance of LspAi against both Copilot and Naive.

Copilot Workflow Setting. For the Copilot implementation, we used Copilot Language Server SDK [1] and invoked the panel completion API with templated prompts. We tried our best to simulate a realistic usage scenario. Specifically, we followed developer recommendations for unit testing [2] and adopted the well-known workflow [35] of Copilot. For large-scale experiments, we automated the unit test generation process by first opening the code file containing the target method, then prompting Copilot to generate the unit tests, and saving them using standard unit test naming conventions.

Model Selection. To demonstrate the effectiveness of LspAi, we evaluate it using language models with different architectures and sizes. For architecture, we include both transformer-based models like the GPT series [7], and mixture-of-experts (MoE) models like DeepSeek [24] and Mistral [20]. For size variation, we test with GPT-4o, GPT-4o-m (GPT-4o-mini), and DS-V3 (Deepseek-V3).

Real-world Projects. For a fair evaluation, we selected real-world projects from either (1) benchmarks used in previous research or (2) projects that are widely adopted by the community. As Table 2 shows, we selected two projects for each programming language. For Java code bases, we choose Commons-CLI [8] and Commons-CSV [9], which are commonly selected as benchmarks by previous research [39, 41] of unit test domain. For Golang code bases, we chose Logrus [34], and Cobra [11], because they are also selected by previous research [38] for evaluating unit test performance. Finally, for Python, we chose Black [10], which is also widely selected as a benchmark by previous research [21, 25]. We chose Crawl4AI [36] to test the ability of LLM's code generation ability against a relatively new project that is not included in LLM training datasets. The Crawl4AI project is one of the most trending projects on GitHub with 24.6k stars, released in May 2024, which is well-developed while surely unseen in the training dataset of LLMs. In terms of evaluation scope, Black includes 472 focal methods, Crawl4AI has 377, Cobra has 155, Commons-CLI contains 140, Commons-CSV

[1] For detailed prompt snippets used by LspAi, refer to https://github.com/Gwihwan-Go/LSPAI/tree/fse-industry/src/promptBuilder.ts

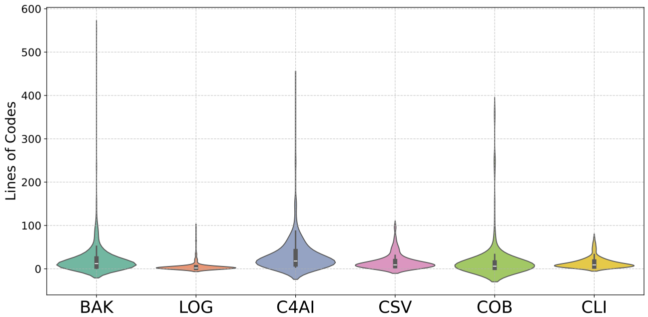

Figure 3: Focal Method Stastics for Real-World Projects.

Table 3: Comparative experiment results on line coverage and valid rate.

	Model	Line Coverage LspAi	Line Coverage Naive	Line Coverage Copilot	Valid Rate LspAi	Valid Rate Naive	Valid Rate Copilot
CLI	GPT-4o	**66.99%**	39.92%	24.46%	**77.33%**	54.67%	26.00%
	GPT-4o-m	**58.55%**	18.78%	-	**59.33%**	24.00%	-
	DS-V3	**61.92%**	47.65%	-	**79.33%**	43.33%	-
CSV	GPT-4o	**50.53%**	35.62%	23.94%	**55.71%**	31.43%	14.86%
	GPT-4o-m	**42.25%**	17.56%	-	**38.57%**	6.43%	-
	DS-V3	**68.26%**	41.01%	-	**43.57%**	10.71%	-
LOG	GPT-4o	**32.95%**	1.16%	1.86%	**21.74%**	4.29%	2.86%
	GPT-4o-m	**30.39%**	2.78%	-	**14.49%**	4.29%	-
	DS-V3	**54.76%**	34.11%	-	**40.58%**	21.43%	-
COB	GPT-4o	**15.75%**	0.20%	5.39%	**17.53%**	10.32%	7.10%
	GPT-4o-m	**7.52%**	2.34%	-	**11.04%**	8.39%	-
	DS-V3	**53.76%**	16.36%	-	**45.54%**	15.65%	-
BAK	GPT-4o	**50.44%**	48.01%	26.95%	57.60%	**47.35%**	81.28%
	GPT-4o-m	**38.62%**	37.28%	-	51.94%	**59.55%**	-
	DS-V3	**41.18%**	40.24%	-	**71.76%**	67.20%	-
C4AI	GPT-4o	**34.42%**	34.53%	20.10%	**56.52%**	53.07%	84.58%
	GPT-4o-m	**37.36%**	3%	-	52.78%	**62.59%**	-
	DS-V3	**40.70%**	39.69%	-	**67.90%**	62.86%	-
	Total	**38.69%**	23.20%	-	**44.66%**	30.92%	-

includes 74, and Logrus has 70. The distribution of method sizes and method counts for the Python projects is shown in Figure 3.

Environment. We adopt the default temperature and generation settings for all LLMs. We conducted our evaluation on a machine equipped with an AMD EPYC 7763 CPU (2.25GHz) with 128 cores and 8 NVIDIA GPU (V100-32G), running Ubuntu 22.04 LTS.

4.2 Comparative Experiment

This section evaluates the performance of LspAi using two metrics: line coverage and valid rate, where a test script is considered valid if it runs without any execution errors. Assertion failures are not treated as errors. As shown in Table 3, LspAi significantly improves both line coverage and valid rate across multiple programming languages, projects, and LLMs. To better understand these results, we provide a detailed analysis. Since each programming language shows slightly different trends, we analyze them separately.

Java. On average, LspAi improves line coverage by 122% and valid rate by 149% compared to Naive, and by 145% and 262% respectively compared to Copilot. The primary reason for the increased coverage is the dependency retrieval-guided unit test generation, which effectively utilizes summarized dependency information of classes and methods to cover diverse edge cases. For the valid rate, LspAi achieves substantial improvements by 149% compared to Naive, and 262% compared to Copilot.This is because of the highly structured Java program's nature, which allows most errors to be detected before compilation through LSP. This is particularly evident in the substantial valid rate improvement observed in Java projects (*e.g.,* CLI and CSV) compared to Naive and Copilot. These results highlight the strength of combining static analysis with LLMs in strongly typed languages like Java.

Golang. For Golang projects, LspAi shows the most impressive improvement in unit test code generation. Specifically, it increases line coverage and valid rate compared to Naive by 2,003% and 171%, respectively, and compared to Copilot by 931% in line coverage and 403% in valid rate. For Golang-specific tasks, we found that GPT-series LLMs often generate invalid unit test code due to simple mistakes. LspAi addresses this by detecting and correcting such errors, leading to a significant improvement in the valid rate, which in turn boosts line coverage. For instance, during our experiments, we observed that LLMs frequently "redeclare" objects already defined in the source code, resulting in invalid test code and low valid rates in the LOG and COB projects (averaging 10.00% and 11.45% for Naive). By leveraging *LSP-Guided Diagnosis*, LspAi detects and resolves this issue, increasing the valid rate from 10.00% to 25.60% for the LOG project and from 11.45% to 24.70% for the COB project. On the other hand, we observed that DeepSeek performs comparably well in generating code for Golang projects compared to GPT-series LLMs. When combined with LspAi, it is particularly effective at generating and fixing grammatical errors in Golang test code. The high valid rate and line coverage achieved by the DeepSeek-guided LspAi support this observation. We assume this may be because DeepSeek was pre-trained at the repository level with up-to-16K-token windows and a fill-in-the-middle objective [19]. This larger context allows it to see existing imports and identifiers and reuse them instead of redefining them. Overall, the rapid improvement in both line coverage and valid rate demonstrates that LspAi can significantly enhance the quality of generated test code, especially for programming languages where LLMs typically struggle.

Python. For Python projects such as BAK and C4AI, LspAi achieved an averaged modest increase in line coverage of 6.02%, but a decrease in the valid rate of 1.41% on average compared to Naive, and 104% on line coverage but 33.17% decrease in valid rate. There are two different trend in improvement compared to Java and Golnag Project. First, the improvement in line coverage is relatively low. This is attributed to two factors. First, LLMs are proficient in Python, as evidenced by Naive attaining the highest average valid rate of 59.99% compared to others. Second, Python's dynamic nature makes LspAi difficult to detect errors before code execution, preventing LSP from identifying issues early and thereby degrading performance. Consequently, LspAi in some cases decreased the valid rate for BAK and C4AI projects, indicating that LSP-guided diagnosis may not fully capture potential errors in Python tasks. Nonetheless, LspAi successfully increased line coverage by generating unit tests despite of lower valid rate, this indicates the retrieved dependency information successfully lead LLMs to generate effective unit tests. The second different trend is that the unit tests generated by Copilot generally shows the highest valid rate, but the lowest coverage. This is because, in our experiment setting, Copilot prone to only

Table 4: Time and Token Usage by LspAi.

	Time (milliseconds)					Token		
	Retrieval	Diagnosis	GEN	FIX	Total	GEN	FIX	Total
Java	38,578	19,168	11,669	16,194	85,789	1,541	2,866	4,407
Golang	5,925	5,337	11,177	26,938	49,377	1,150	4,885	6,035
Python	98,533	22,033	13,203	4,604	138,373	1,460	510	1,970
Averaged	47,738	15,512	12,016	15,912	91,179	1,383	2,753	4,137

generate assertion codes rather than generate complicate code logic. For example, as shown in Listing 1, the overall generation policy is a multiple lines of assertion codes under unit test class. This contributes the least error, but not able to maximize the coverage of codes. Overall, This shows that collected dependency information by LspAi generated effective unit tests that cover more edge cases, resulting in more reliable test cases with higher coverage.

Listing 1: Python Unit Test Code Generated by Copilot

```
class Test__init___handle_charref_1_test(unittest.TestCase):
    def test_handle_charref_init(self):
        self.assertTrue(hasattr(handle_charref, "__name__"), "...")
        self.assertTrue(hasattr(HTMLParser, "__name__"), "...")
        self.assertTrue(hasattr(HTMLParserError, "__name__"), "...")
```

4.3 Breakdown of LspAi

To evaluate the practical applicability of LspAi in real-world use cases, we measured its time and token consumption for unit test generation. We put the averaged figure classified by programming languages. The statistics were collected using a maximum of five fixing attempts per focal method, with LspAi accessing an LLM via API requests. The evaluation was conducted on the same projects listed in Table 2, and the number of methods is shown in Figure 3. The final results represent the average time for each stage per focal method. All experiments of Table 4 used GPT-4o as the LLM, accessed via API.

As shown in Table 4, LspAi takes approximately 91 seconds and consumes 4,137 tokens on average to generate and refine unit tests. For Golang projects, the utility of LspAi is particularly impressive, as it requires an acceptable range of resources (49 seconds and 6,035 tokens on average) while delivering an 20× improvement in line coverage. This demonstrates the efficiency and effectiveness of LspAi in languages like Golang, where LLMs are less proficient, making automated unit test generation particularly effective. In contrast, LspAi is less ideal for Python projects. While time and token usage remains within acceptable ranges, the modest improvement of approximately 4% in line coverage does not justify the resource expenditure of 2 minute and approximately 2,000 tokens per focal method unless additional performance improvements in test quality can be ensured. This makes LspAi less practical for Python projects compared to its strong performance in other languages. Overall, LspAi demonstrates efficient and scalable performance across a variety of programming languages, but it excels most in contexts where LLMs traditionally underperform, such as with Golang. The results highlight LspAi's potential to bridge performance gaps in automated unit test generation for less-supported languages while maintaining acceptable resource usage for real-world applications.

5 Lessons Learned

In this section, we introduce some lessons learned during building tool for multi-language unit test generation and applying the tool to development environment in Tecent Ltd.

Alignment between Research and Industrial Needs. We identified a significant gap between academic research and industrial requirements in the domain of unit test generation. While academic efforts predominantly aim for high code coverage, they often overlook practical aspects essential for real-world usage. From our industry practice, developers urgently need a lightweight tool that can generate unit tests without whole-project compilation. Additionally, academic research has largely focused on specific programming languages such as Python and Java, leaving a gap in support for other languages. For example, Golang is widely adopted in many industries, yet few academic studies have addressed unit test generation for Golang. LspAi was developed to bridge these gaps. Although it may not achieve the same level of code coverage as established academic tools like EvoSuite [12], LspAi is designed with industrial applicability in mind, aiming for widespread use in industry scenarios.

Varying Language Proficiency of LLMs. Our experimental analysis revealed that LLMs exhibit varying levels of proficiency across different programming languages, directly impacting the effectiveness of LspAi. Specifically, LLMs frequently make errors when generating Golang code, which affects the quality of the generated unit tests. These findings highlight the necessity for LLM-integrated tools to adapt their strategies to the specific demands and complexities of each programming language, thereby maximizing their utility and effectiveness.

Demands for Better Integration Methods. This work opens several promising avenues for future research in multi-language unit test generation through the LSP. Currently, LspAi employs prompt engineering, a low-cost but limited method for harnessing the full potential of LLMs. A more sophisticated integration method is needed to build a retrieval system that can fully exploit the capabilities of LLMs. Besides, the integration of additional language server functionalities, such as code intelligence and code action recommendations, could further enhance the accuracy and reliability of generated unit tests. In our industrial practice, we found that developers often require a more comprehensive approach to make the generated unit tests more reliable and useful.

6 Conclusion

We introduced LspAi, a practical real-time unit test generation tool that leverages LLMs and integrates static analysis through the LSP to support multi-language codebases. LspAi addresses the critical gap in existing research by enabling seamless unit test generation across diverse programming languages without the need for project-wide compilation, thereby facilitating concurrent test creation alongside code development. Implemented as an IDE plugin, LspAi simplifies adoption for developers by requiring only the installation of appropriate language analysis tools. Our comprehensive evaluation of Java, Python, and Golang projects demonstrated that LspAi consistently enhances both line coverage and valid rate compared to baseline approaches.

Data-Availability Statement

All data and materials supporting the findings of this study are openly available at Zenodo [16].

References

[1] copilot-language-server-release. GitHub repository, 2025. Accessed: 2025-04-10.
[2] Question about configuring copilot's language model. GitHub Issue, 2025. Accessed: 2025-04-10.
[3] Juan Altmayer Pizzorno and Emery D Berger. Coverup: Coverage-guided llm-based test generation. *arXiv e-prints*, pages arXiv–2403, 2024.
[4] Yiran Cheng, Lwin Khin Shar, Ting Zhang, Shouguo Yang, Chaopeng Dong, David Lo, Shichao Lv, Zhiqiang Shi, and Limin Sun. Llm-enhanced static analysis for precise identification of vulnerable oss versions. *arXiv preprint arXiv:2408.07321*, 2024.
[5] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. Effective test generation using pre-trained large language models and mutation testing. *Inf. Softw. Technol.*, 171(C), July 2024.
[6] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. Leveraging Large Language Models for Enhancing the Understandability of Generated Unit Tests . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 392–404, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.
[7] OpenAI et al. Gpt-4 technical report, 2024.
[8] Apache Software Foundation. Apache commons cli, 2025. Accessed: 2025-01-15.
[9] Apache Software Foundation. Apache commons csv, 2025. Accessed: 2025-01-15.
[10] PSF (Python Software Foundation). Black, 2025. Accessed: 2025-01-15.
[11] Steve Francia. Cobra. https://github.com/spf13/cobra, 2013. Accessed: 2025-01-15.
[12] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.
[13] Gordon Fraser and Andrea Arcuri. A large scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):8, 2014.
[14] Nat Friedman. Introducing github copilot: Your ai pair programmer, 2021. Accessed: 2024-12-20.
[15] Tianyu Gao, Adam Fisch, and Danqi Chen. Making pre-trained language models better few-shot learners. *arXiv preprint arXiv:2012.15723*, 2020.
[16] Gwihwan Go. Lspai. https://zenodo.org/records/15206535, April 2025. Presented at the FSE Industry conference in Trondheim, Norway.
[17] Gwihwan Go, Chijin Zhou, Quan Zhang, Xiazijian Zou, Heyuan Shi, and Yu Jiang. Towards more complete constraints for deep learning library testing via complementary set guided refinement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1338–1350, New York, NY, USA, 2024. Association for Computing Machinery.
[18] Nuno M Guerreiro, Duarte Alves, Jonas Waldendorf, Barry Haddow, Alexandra Birch, Pierre Colombo, and André FT Martins. Hallucinations in large multilingual translation models. *arXiv preprint arXiv:2303.16104*, 2023.
[19] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.
[20] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024.
[21] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha 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)*, pages 919–931. IEEE, 2023.
[22] Ziyang Li, Saikat Dutta, and Mayur Naik. Llm-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238*, 2024.
[23] Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. Recovering fitness gradients for interprocedural boolean flags in search-based testing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 440–451, 2020.
[24] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.
[25] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172, 2022.
[26] Microsoft. Best practices for using github copilot, 2024. Accessed: 2024-12-20.
[27] Microsoft. Language server protocol, 2024. Accessed: 2024-12-24.
[28] Microsoft. Syntax highlight guide, 2024. Accessed: 2024-12-20.
[29] Chao Ni, Xiaoya Wang, Liushan Chen, Dehai Zhao, Zhengong Cai, Shaohua Wang, and Xiaohu Yang. Casmodatest: A cascaded and model-agnostic self-directed framework for unit test generation. *arXiv preprint arXiv:2406.15743*, 2024.
[30] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.
[31] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering*, 1(FSE):951–971, 2024.
[32] Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. Testspark: Intellij idea's ultimate test generation companion. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 30–34, 2024.
[33] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. Using large language models to generate junit tests: An empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, page 313–322, New York, NY, USA, 2024. Association for Computing Machinery.
[34] Sirupsen. Logrus, 2025. Accessed: 2025-01-15.
[35] Parth Thakkar. Copilot explorer, 2022. Accessed: 2024-12-20.
[36] UncleCode. Crawl4ai, 2025. Accessed: 2025-01-15.
[37] Han Wang, Han Hu, Chunyang Chen, and Burak Turhan. Chat-like asserts prediction with the support of large language model. *arXiv preprint arXiv:2407.21429*, 2024.
[38] Siwei Wang, Xue Mao, Ziguang Cao, Yujun Gao, Qucheng Shen, and Chao Peng. Nxtunit: Automated unit test generation for go. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, page 176–179, New York, NY, USA, 2023. Association for Computing Machinery.
[39] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. Hits: High-coverage llm-based unit test generation via method slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1258–1268, 2024.
[40] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.
[41] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. Chatunitest: a chatgpt-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764*, 2023.
[42] Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. Enhancing llm-based test generation for hard-to-cover branches via static analysis. *arXiv preprint arXiv:2404.04966*, 2024.
[43] Zhichao Zhou, Yutian Tang, Yun Lin, and Jingzhu He. An llm-based readability measurement for unit tests' context-aware inputs. *arXiv preprint arXiv:2407.21369*, 2024.

Received 2025-01-12; accepted 2025-03-26