

SCOUT: A Scenario-Guided Unit Test Generation Tool for Compact Open-Source LLMs

Myungho Lee

R&D Center
Samsung SDS
Seoul, Republic of Korea
frango.lee@samsung.com

Yoon-Chan Jhi

R&D Center
Samsung SDS
Seoul, Republic of Korea
yoontan.jhi@samsung.com

Seokhyeon Moon

R&D Center
Samsung SDS
Seoul, Republic of Korea
shyeon.mun@samsung.com

Hakjoo Oh[✉]

Korea University
Seoul, Republic of Korea
hakjoo_oh@korea.ac.kr

Abstract

Large language models (LLMs) have shown promising performance in automated unit test generation, but existing approaches often rely on large proprietary or reasoning-oriented models, limiting their practicality for users with limited computational resources. We present SCOUT, a scenario-guided unit test generation tool designed for compact open-source LLMs. Rather than requiring LLMs to reason over raw source code, SCOUT extracts program information through static analysis and progressively provides concise, task-specific contexts for specialized reasoning tasks, including scenario construction and unit test generation. The tool further validates and refines generated tests using compilation, execution, and coverage feedback. Built on an extensible multi-agent architecture, SCOUT provides a fully automated workflow that enables practical unit test generation with locally deployed open-source LLMs on commodity hardware.

CCS Concepts: • Software and its engineering → Software testing and debugging.

Keywords: Unit Testing, Automated Test Generation, Large Language Models

ACM Reference Format:

Myungho Lee, Seokhyeon Moon, Yoon-Chan Jhi, and Hakjoo Oh. 2026. SCOUT: A Scenario-Guided Unit Test Generation Tool for Compact Open-Source LLMs. In *Companion Proceedings of the 2026 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '26)*, October 4–9, 2026, Oakland, CA, USA. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3837729.3840501>



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

SPLASH Companion '26, Oakland, CA, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2910-2/2026/10

<https://doi.org/10.1145/3837729.3840501>

1 Introduction

Unit testing is essential for software quality and regression prevention [3]. Recent advances in large language models (LLMs) have enabled automated generation of effective, human-readable unit tests [4, 5, 8?]. However, the strongest existing approaches often rely on large proprietary or reasoning models [1, 6, 7]. Their effectiveness can degrade with compact open-source LLMs, which are less capable of simultaneously understanding large code contexts, inferring testing objectives, and repairing invalid tests.

To address this challenge, we present SCOUT, a scenario-guided unit test generation tool designed to maximize the effectiveness of compact open-source LLMs. The central idea of SCOUT is to reformulate unit test generation as a sequence of compact, specialized reasoning tasks rather than a single large generation task. Instead of directly forwarding raw program artifacts to the LLM, SCOUT continuously distills program information, testing objectives, compilation results, and coverage information into concise task-specific contexts. These contexts are then consumed by specialized agents that focus on scenario construction, test generation, validation, and feedback-driven refinement.

In SCOUT (a **SC**enario-**O**-guided Unit test generation **T**ool), a testing scenario is a compact representation of a test intent. It describes the behavior to be exercised, relevant preconditions, expected outputs, and the program elements that should be targeted. Scenarios serve as an intermediate abstraction between static program information and executable unit tests. By making testing objectives explicit before code generation, SCOUT allows LLMs to reason over smaller and more structured inputs while still preserving the semantic information needed to generate meaningful tests.

To summarize, SCOUT provides three key features.

- **Context-Compact LLM Workflow:** SCOUT offloads program inspection and artifact processing to lightweight analysis components, invoking LLMs only for

language-intensive reasoning tasks. Each LLM invocation receives a concise context tailored to its specific role, reducing unnecessary reasoning burden on compact models.

- **Scenario-Guided Test Generation:** SCOUT introduces testing scenarios as intermediate representations that bridge program information and executable unit tests. By explicitly representing test intent before code generation, the tool helps LLMs generate tests that are more focused, interpretable, and aligned with target program behavior.
- **Feedback-Driven Automated Refinement:** SCOUT automates the complete generation loop, including scenario construction, test synthesis, compilation, execution, repair, and coverage-guided refinement. Compilation errors, runtime failures, and uncovered code elements are summarized into structured feedback for subsequent generation steps.

Intended Users. SCOUT is intended for developers and researchers who wish to perform automated unit test generation using locally deployable open-source LLMs on commodity hardware. Unlike approaches that rely on large proprietary models or require substantial manual intervention, SCOUT provides an end-to-end workflow that can operate under practical resource constraints. Its modular multi-agent architecture also allows new components, such as static analysis techniques, validation strategies, and refinement agents, to be integrated with minimal engineering effort.

Contributions. In summary, we contribute:

- We introduce a scenario-guided generation workflow that decomposes unit test generation into specialized reasoning stages connected by compact intermediate representations.
- We present SCOUT, a fully automated and extensible implementation that integrates static analysis, LLM-based scenario construction, unit test generation, validation, repair, and coverage-guided refinement.
- We evaluate SCOUT on real-world Java projects and show that its staged, context-compact workflow improves structural coverage when using compact open-source LLMs.

2 Design and Implementation

2.1 System Architecture

Figure 1 presents the overall architecture of SCOUT. The tool adopts a modular multi-agent architecture composed of four specialized agents that collaboratively generate unit tests for a target Java project. The agents interact with a LLM and are supported by static and dynamic analysis components that continuously provide contextual information and runtime feedback throughout the generation process.

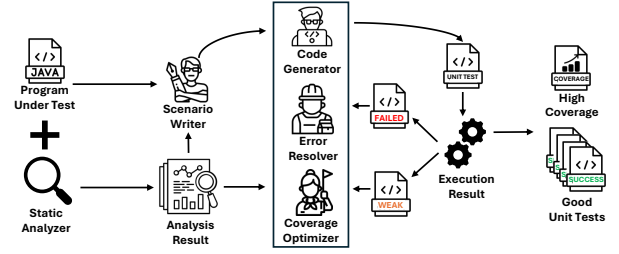


Figure 1. Workflow of our approach.

```

SCOUT Status
-----
LLM      : Qwen/Qwen2.5-3B-Instruct
Phase    : DEFAULT
Mode     : multithreading
Threads  : 2 / 2 (class 1, method 2)
Coverage : report on
Project  : /home/frango/.bench/threeten-extra

Methods  : 2 / 1074 [.....] 0%
Time     : 00:00:05 elapsed | ETA 00:51:45

Valid methods : 0
Full line cov : 0
Exported tests : 0
Compile errors : 0
Runtime errors : 2
Timeouts      : 0
LLM attempts  : 10
LLM successes  : 0

...ra/DEFAULT/Qwen-Qwen2.5-3B-Instruct/20260626-053023-312

```

Figure 2. CLI interface of SCOUT, showing the current execution status, including the selected LLM, elapsed time, and the numbers of valid and invalid unit tests generated during execution.

SCOUT is built upon ChatUniTest [1], using its Java implementation. Since the original implementation is designed primarily as a Maven plugin, we refactored it into a standalone Java framework with a modular architecture. This redesign decouples the unit test generation pipeline from project-specific build environments, allowing the tool to be used independently and making it easier to integrate new unit test generation techniques. Furthermore, the multi-agent framework for SCOUT is also modularized, enabling developers to readily add, replace, or customize individual agents with minimal implementation effort.

The workflow starts with the *Static Analyzer*, which extracts project information and program information from the target software. The *Scenario Writer* uses this information to derive testing scenarios that guide the *Code Generator* in producing unit tests with a LLM. The generated tests are validated through compilation, execution, and coverage analysis. Rather than directly forwarding the runtime outputs to the LLM, SCOUT first distills compilation diagnostics and coverage reports into compact, task-specific contexts, which can alleviate the burdens on LLMs. The *Error Resolver* and

Coverage Optimizer then use this refined feedback to iteratively improve test quality. This iterative workflow enables SCOUT to generate effective unit tests with compact open-source LLMs while remaining extensible to future analysis techniques and agents.

2.2 Core Components

SCOUT adopts a modular multi-agent architecture that combines lightweight program analysis with LLM-based reasoning for automated unit test generation. Rather than relying on LLMs throughout the entire pipeline, SCOUT employs static and dynamic analyses to extract program information and execution feedback, invoking LLMs only for tasks that require semantic reasoning, such as generating testing scenarios, synthesizing test code, repairing invalid tests, and rewriting test codes for higher coverage. This design reduces unnecessary LLM interactions while improving both the validity and coverage of generated unit tests.

2.2.1 Analyzers. To support the multi-agent workflow, SCOUT incorporates static and dynamic analyzers that provide contextual information throughout the whole generation process. The static analyzer extracts structural information before test generation, while the dynamic analyzer collects execution feedback after tests are generated. These analyzers continuously exchange structural and runtime information, producing refined feedback that enables the LLM agents to iteratively improve both the validity and coverage of generated unit tests.

Static Analysis. The static analyzer extracts structural information from the target Java project, including method signatures, field declarations, dependency relationships. This information is provided to *Scenario Writer* and *Code Generator* to construct accurate prompts and generate valid unit tests. If full coverage is not achieved, the analyzer further performs lightweight dependency analysis on uncovered code regions. The resulting dependency information highlights relevant parameters, fields, and method invocations, which are later used by *Coverage Optimizer* to generate additional tests for unexplored branches.

Dynamic Analysis. The dynamic analyzer compiles and executes generated unit tests, collecting execution outcomes and structural coverage information. Rather than forwarding raw execution results, it analyzes compilation failures, runtime exceptions, and coverage to extract actionable feedback for subsequent agents.

For invalid tests, the analyzer identifies the relevant error locations and summarizes the underlying causes before providing the information to *Error Resolver*, enabling targeted repair. For coverage improvement, it combines runtime coverage information with dependency information obtained from static analysis to identify unexplored execution paths

and the program elements related to them. This refined feedback is then supplied to *Coverage Optimizer*, allowing it to generate additional tests more effectively.

2.2.2 Agents. LLM agents perform the reasoning tasks throughout the test generation pipeline. Guided by the refined structural and runtime information produced by the analyzers, each agent is responsible for a specific stage of the workflow, including scenario generation, test synthesis, error repair, and coverage improvement. This separation allows SCOUT to invoke LLMs only where semantic reasoning is required while relying on lightweight program analyses for contextual information.

Scenario Writer. *Scenario Writer* derives testing scenarios from multiple levels of project context, including project information, class information, and target method source code. The resulting scenarios describe representative execution paths, input conditions, and expected behaviors of the target method. By organizing these behaviors into explicit testing objectives, the scenarios provide structured, high-level guidance that helps the subsequent test generation focus on diverse program behaviors rather than reasoning directly over the entire program context.

Code Generator. Using the generated scenarios together with method source code, dependency information, method signatures, and object instantiation constraints, *Code Generator* synthesizes executable JUnit test cases. The additional contextual information helps the LLM generate valid and comprehensive tests while reducing compilation failures.

Error Resolver. *Error Resolver* improves test validity when generated tests fail to compile or execute. It first applies lightweight rule-based repairs for common syntactic issues and missing imports. If errors persist, it invokes the LLM with error-specific feedback to repair the generated tests.

Coverage Optimizer. *Coverage Optimizer* leverages the refined feedback produced by the analyzers to improve structural coverage. Using uncovered branches together with their associated dependency information, it refines prompts and requests additional tests that target previously unexplored execution paths.

2.3 Command-Line Interface

Users can use SCOUT with the following command-line interface (CLI) command:

```
1 java -jar scout.jar -project <target_project>
2   -url <api_url> -model <model_name>
3   -phase <phase_name>
4   ...
```

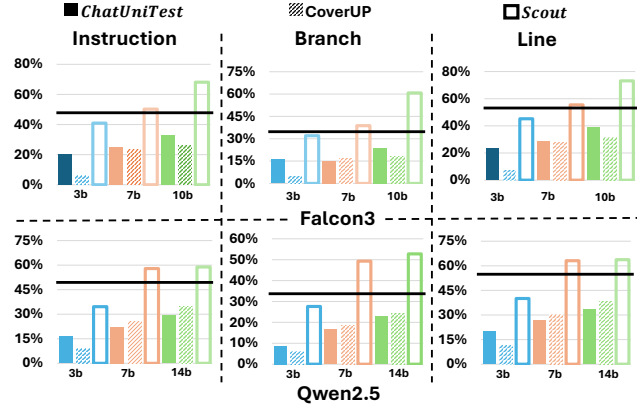


Figure 3. Average coverage on threeten-extra project over three trials.

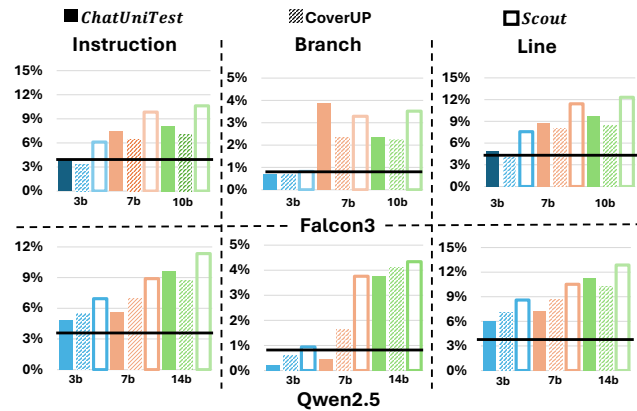


Figure 4. Average coverage on the industrial project over three trials.

`api_url` specifies the endpoint hosting the LLM, which can be either a public or local URL. `model_name` specifies the LLM to use, while `target_project` denotes the root directory of the target project. The `phase_name` option specifies the unit test generation strategy. As shown in Figure 2, after execution, SCOUT displays a status window for monitoring the test generation process. It shows the current generation phase, selected LLM, and estimated remaining execution time. SCOUT also provides optional arguments for multithreading, targeting a specific class, and other advanced configurations. Further setup and usage instructions are provided in the README of the project repository.

3 Evaluation

We evaluated the effectiveness of SCOUT on both open-source and industrial Java projects. Since SCOUT is designed for developers with limited computational resources, our evaluation focuses on compact open-source LLMs. In particular, we investigate whether the scenario-guided generation strategy enables such models to achieve competitive test-generation performance without relying on substantially larger LLMs.

Evaluation Setting. To evaluate SCOUT, we first selected the two highest-ranked officially released models with approximately 7 billion parameters on the Hugging Face Open LLM Leaderboard¹, namely *Falcon3* and *Qwen2.5*. We focus on 7B-scale models because they provide a practical balance between performance and computational cost and can be deployed more readily in resource-constrained environments. For each model family, we additionally included the next smaller and next larger officially released variants, resulting in a total of six LLMs. This selection allows us to examine whether the effectiveness of SCOUT remains consistent across different model families and parameter scales rather than depending on a particular LLM. All results are averaged over three trials to mitigate the randomness of LLMs.

As baselines, we selected ChatUniTest and CoverUP, both of which are implemented on top of the same *ChatUniTest-Core* framework as SCOUT. Since these three tools share the same underlying infrastructure, the comparison primarily highlights the effectiveness of their respective test generation strategies rather than implementation-specific differences. In particular, CoverUP provides an informative comparison because it also uses coverage feedback to iteratively improve generated tests, but does not explicitly construct testing scenarios before generation.

Open-Source Java Project. We first evaluated the tools on *threeten-extra*, an open-source Java project containing 66 Java classes, 11,799 lines of code, and 1,074 target methods for unit test generation.

Figure 3 compares the instruction, branch, and line coverage achieved by the evaluated tools using Falcon3 and Qwen2.5 of varying parameter sizes, measured using JaCoCo. Solid, striped, and blank box plots correspond to the tools ChatUniTest, CoverUP, and SCOUT, respectively, while the horizontal solid line indicates the performance of ChatUniTest using the *gpt-oss-20B* model.

As shown in Figure 3, SCOUT consistently outperformed both ChatUniTest and CoverUP using the same LLM across all three coverage metrics. More importantly, SCOUT enabled substantially smaller models to achieve comparable or higher coverage than larger models used by the baselines. For example, Falcon3-7B with SCOUT achieved instruction, branch, and line coverage of 50.3%, 39.7%, and 55.3%, respectively, slightly exceeding the 48.3%, 39.0%, and 54.3% achieved by ChatUniTest with the 20B *gpt-oss-20B*. This result is notable not for the absolute coverage gain, but because SCOUT achieves it with a model nearly three times smaller. The consistent improvement over CoverUP, which also employs iterative coverage feedback, further supports the benefit of explicit scenario guidance.

¹https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard

Table 1. Number of fully covered methods in threeten-extra. N denotes the total number of methods in each branch group.

#Branch (N)	Tool	Falcon3			Qwen2.5		
		3B	7B	10B	3B	7B	14B
0 (633)	ChatUniTest	191	135	282	117	178	248
	CoverUP	41	160	213	55	179	342
	SCOUT	422	301	551	278	423	470
1 – 5 (315)	ChatUniTest	27	56	53	12	49	63
	CoverUP	14	56	49	16	61	134
	SCOUT	124	96	194	85	153	194
> 5 (126)	ChatUniTest	1	6	12	0	9	10
	CoverUP	0	6	7	0	7	12
	SCOUT	8	13	32	10	21	31

```

1 public boolean equals(@Nullable Object obj) {
2     if (this == obj) return true;
3     if (obj instanceof YearHalf) {
4         YearHalf other = (YearHalf) obj;
5         return year == other.year && half == other.half;
6     }
7     return false;
8 }

```

Figure 5. equals() method from YearHalf class.

```

1 SCENARIO: Testing the equals method
2
3 - Given a YearHalf instance 'yearHalf1' created with '2023-05'
4 - When compared with another YearHalf instance yearHalf1
5 - Then the result is true
6
7 - Given a YearHalf instance 'yearHalf1' created with 2023-05
8 - When compared with a null value
9 - The the result is false (null comparison)
10 ...

```

Figure 6. Examples of scenarios generated for Figure 5.

Branch-Level Analysis. Table 1 reports fully covered methods grouped by their number of branches. Overall, SCOUT fully covers more methods than the baselines across most LLMs and branch groups. The difference remains evident for methods with more complex control flow. Among the 126 methods with more than five branches, Falcon3-10B with SCOUT fully covers 32, compared with 12 and 7 for ChatUniTest and CoverUP, respectively. Since fully covering such methods requires tests to exercise multiple control-flow paths, these results suggest that scenario guidance helps the LLM identify and exercise diverse program behaviors rather than merely increasing coverage on simpler methods.

Effectiveness of Scenario. Figure 5 shows the target method, while Figure 6 presents a subset of the scenarios generated by SCOUT for this method. These scenarios provide explicit testing objectives that guide subsequent test generation, enabling SCOUT to fully cover the method even with 3B models. In contrast, despite the relatively simple control flow, neither ChatUniTest nor CoverUP achieves full coverage even with the larger Falcon3-10B and Qwen2.5-14B models. This example illustrates how scenario guidance helps compact LLMs exercise behaviors that may otherwise be overlooked.

Token Consumption. SCOUT incurs additional token consumption due to its multi-stage reasoning process. For example, Falcon3-3B with SCOUT consumed 82M tokens, whereas Falcon3-10B with ChatUniTest and CoverUP consumed 24M and 45M tokens, respectively. Despite this higher token cost, SCOUT with the 3B model achieved higher coverage than both tools using the substantially larger 10B model. This suggests a practical trade-off for resource-constrained environments: additional inference-time reasoning can enable compact models to achieve better test generation performance without requiring the GPU memory needed to deploy larger models.

Industrial Java Project. We additionally evaluated a proprietary cryptographic library developed and used within our organization, which has never been open publicly. This library is utilized for critical security functions, such as database encryption, and is certified under ISO/IEC 15408 (CC certification) [2]. As it is a multi-module project, we selected one representative module consisting of 95 classes, 255 target methods, and 5,534 lines of code.

Similar to Figure 3, Figure 4 presents coverage achieved by the evaluated tools on the industrial project. Since the industrial project was not publicly available and therefore unlikely to have appeared in the open-sourced LLMs’ training data, all three tools with LLMs showed poor performance on achieving coverage compared to Figure 3. However, SCOUT still consistently achieved the highest coverage among the evaluated approaches across all three coverage metrics. These results indicate that the scenario-guided workflow generalizes beyond open-source software and remains effective on real-world industrial systems.

4 Conclusion

We presented SCOUT, a scenario-guided unit test generation tool for compact open-source LLMs. By combining scenario construction with feedback-driven refinement, SCOUT enables compact LLMs to generate tests with improved structural coverage. Our evaluation on open-source and industrial Java projects shows that SCOUT consistently outperforms other tools using the same LLMs and often enables smaller models to surpass substantially larger ones. We hope SCOUT provides a practical solution for LLM-based unit test generation under limited GPU resources.

5 Tool Availability

The source code of SCOUT is publicly available on GitHub. The repository includes the project overview, installation instructions, usage documentation, and examples. A demonstration video of the current version is also available online.

- **Project Repository:**

<https://github.com/samsungsds-research-papers/SCOUT>

- **Demonstration Video:** <https://youtu.be/xrrnUOMxcEc>

References

- [1] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 572–576. doi:10.1145/3663529.3663801
- [2] ISO/IEC. 2009. ISO/IEC 15408-1:2009. <https://www.iso.org/standard/50341.html>.
- [3] Vladimir Khorikov. 2020. *Unit testing principles, practices, and patterns*. Simon and Schuster.
- [4] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. 2025. LlamaRestTest: Effective REST API Testing with Small Language Models. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 465–488. doi:10.1145/3715737
- [5] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. 2024. Aster: Natural and multi-language unit test generation with llms. *arXiv preprint arXiv:2409.03093* (2024).
- [6] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2024. ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1340–1359. doi:10.1109/TSE.2024.3382365
- [7] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1258–1268. doi:10.1145/3691620.3695501
- [8] Chen Yang, Junjie Chen, Bin Lin, Ziqi Wang, and Jianyi Zhou. 2026. Advancing Code Coverage: Incorporating Program Analysis with Large Language Models. *ACM Transactions on Software Engineering and Methodology* 35, 5, Article 118 (2026), 31 pages. doi:10.1145/3748505

Received 2026-06-27; accepted 2026-07-25