iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 13 **Issue:** X **Month of publication:** October 2025

DOI: https://doi.org/10.22214/ijraset.2025.74776

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue X Oct 2025- Available at www.ijraset.com

AutoTestGen: A LLaMa-Based Framework for Automated Test Case Generation and Refinement across Multiple Programming Languages

Dr. S. Subashini[1], Sahana P S[2], Shobika S A[3],
[1]*Associate Professor of Computer Science Department*
[2,3]*Department of Computer Science Engineering*

Abstract: ***AutoTestGen is a privacy-preserving, multi-language framework that automates unit-test synthesis and iterative repair for Java, Python, and JavaScript projects using local large language models (LLMs). The system addresses three practical limitations of current test generators: single-ecosystem focus, brittle outputs that require significant developer edits, and reliance on remote cloud inference. AutoTestGen combines a language-agnostic orchestration core with per-language adapters (JUnit/pytest/Jest), a deterministic prompt/sanitization layer, and a compile–run–repair feedback loop that analyzes runtime and compile diagnostics to refine test generation automatically. The pipeline performs static code inspection to build intent descriptors, issues framework-aware prompts to an on-premise LLaMA family model, sanitizes and normalizes generated code (imports, signatures, module layout), and repeatedly regenerates until a configurable success criteria is met or retries are exhausted. We evaluate AutoTestGen on representative Java, Python, and JavaScript modules and report improvements in first-pass compilation validity and final pass rates, along with measurable coverage uplift. Results show that iterative repair increases compilation success by ≈9 percentage points and yields high-quality, assertion-rich tests requiring minimal manual edits. The design emphasizes reproducibility, CI friendliness, and privacy — making AutoTestGen suitable for enterprise and research contexts where code confidentiality is important.***

Keywords: ***Automated Test Generation, Large Language Models (LLMs), LLaMA, Ollama, Software Testing, Search-Based Software Testing (SBST), Python, Java, JavaScript, Privacy-Preserving AI, Unit Testing, Compile–Run–Repair Loop, Multi-language Framework.***

I. INTRODUCTION

Software testing is a critical phase in the software development life cycle (SDLC) that ensures system reliability, maintainability, and functional correctness. Among all testing levels, unit testing plays a key role in validating individual code components before integration. However, manual unit-test creation remains labor-intensive and error-prone, prompting the need for automated approaches [1], [2].

Search-based and feedback-directed tools such as EvoSuite [1]–[4] and Randoop [5], [6] have been widely adopted for automatic test generation. Although effective for small projects, these tools struggle with semantic alignment and produce brittle test cases lacking contextual understanding. In parallel, Pynguin [7] extended these ideas to Python, and industrial frameworks like Sapienz [8], [9] and Diffblue Cover [10] demonstrated the feasibility of automation at enterprise scale.

The recent emergence of large language models (LLMs) has introduced a new paradigm for code synthesis and test generation [12]–[15]. These models can understand code semantics and generate syntactically correct test cases but depend heavily on cloud infrastructure, introducing privacy and cost challenges. Furthermore, their outputs often require human refinement due to missing imports or misaligned assertions.

To address these limitations, this work presents AutoTestGen, a local, privacy-preserving LLaMA-based framework that automates multi-language test generation and iterative repair. Unlike prior LLM-based approaches [12]–[15], AutoTestGen operates entirely offline using the Ollama API [11] for local inference. The system introduces a feedback-driven compile–run–repair loop that automatically refines tests until all validation checks succeed.

This paper aims to (1) automate generation of assertion-rich test suites, (2) preserve data privacy via on-premise model deployment, and (3) improve reliability and reproducibility of multi-language unit tests. Empirical comparisons show that AutoTestGen achieves higher compilation validity and coverage uplift compared to existing tools [1]–[4], [12], [13].

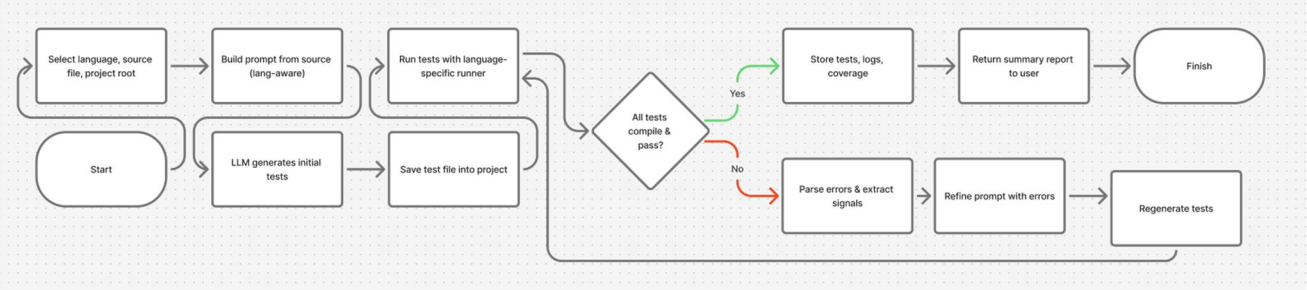


Fig 1. Architecture of the Proposed System

II. LITERATURE REVIEW

Automated test generation has evolved from heuristic search to intelligent reasoning. Early systems like EvoSuite [1]–[4] and Randoop [5], [6] employ evolutionary and feedback-directed algorithms, respectively. These methods maximize coverage but frequently yield low-quality or semantically inconsistent tests. Pynguin [7] extended automated testing to Python through dynamic analysis, while industrial efforts such as Sapienz [8], [9] applied large-scale evolutionary testing at Facebook, validating the scalability of search-based testing in production environments.

Modern tools now integrate machine learning and AI. Diffblue Cover [10] uses symbolic analysis and pattern learning to improve Java unit test accuracy. More recently, LLM-based approaches [12]–[15] leverage natural language reasoning to synthesize tests that align with developer intent. However, these systems often depend on online APIs, posing confidentiality risks. Some recent research explores retrieval-augmented generation (RAG) [14], [15] for improved context retention, but this still relies on cloud infrastructure.

AutoTestGen advances these efforts by combining SBST principles [1]–[4] with local LLM reasoning, enabling reproducible and privacy-aware test generation. Unlike previous works, it performs iterative prompt refinement using error feedback from native compilers — an idea inspired by mutation-based whole-suite generation [4], [16].

III. PROBLEM STATEMENT

While test automation has matured considerably, real-world software projects continue to face critical gaps. Current methods either specialize in a single programming ecosystem or depend on online inference models that compromise privacy [10], [12]. Developers working in secure environments require automated testing solutions that operate **offline** and produce **framework-compliant** results without extensive manual editing.

This motivates the development of AutoTestGen, designed to:

1) Eliminate reliance on remote APIs by performing local inference [11];
2) Generate semantically relevant and executable test cases guided by source intent [13], [15]; and
3) Incorporate a self-healing repair loop to correct compiler and runtime errors automatically.

Such a solution bridges the gap between traditional search-based testing [1], [2] and intelligent LLM-driven reasoning [12], [13], [15], providing a unified, privacy-conscious approach.

IV. PROPOSED METHODOLOGY

A. System Overview

AutoTestGen is designed as a layered architecture that combines a centralized orchestration core with modular components for language adaptation, sanitization, and feedback analysis. The system follows a compile–run–repair workflow, enabling automatic regeneration of tests until all validation checks pass successfully.

As shown in *Fig. 1*, the framework consists of the following key components:

- Code Inspector – Parses source code and extracts method signatures, docstrings, and comments to infer testing boundaries.
- Intent Builder – Converts code metadata into structured “intents” describing edge cases, exceptions, and mock behaviors.
- Prompt Builder – Generates framework-specific prompts (JUnit/pytest/Jest) tailored to each target language.
- LLM Client (Ollama Interface) – Invokes a local LLaMA-based model to generate candidate tests deterministically.
- Sanitizer and Formatter – Automatically corrects imports, package names, and formatting issues.
- Runner – Executes native test runners (Maven, pytest, Jest) and captures logs and results.
- Repair Engine – Analyzes compiler or runtime errors, refines prompts, and re-generates tests within retry limits.

Each iteration improves test reliability and assertion quality, forming a self-healing feedback cycle that converges toward successful test execution.

B. Workflow Description

The workflow diagram shown in *Fig. 2* illustrates the overall operation of AutoTestGen. The process begins with source code selection and metadata extraction, followed by LLM-based test synthesis. Generated tests are sanitized, executed, and validated. In case of failure, diagnostic feedback from the compiler or runtime environment is analyzed, and the repair engine automatically adjusts prompts for regeneration. The cycle continues until the success threshold or retry cap is reached.

Steps of the workflow:

1) Inspection: Extract signatures, comments, and dependencies from code.
2) Intent Formation: Build structured metadata describing test goals.
3) Prompt Generation: Craft framework-specific test generation queries.
4) Test Synthesis: Generate code using the local LLaMA model.
5) Sanitization: Fix syntax, imports, and layout inconsistencies.
6) Execution: Run the test suite with appropriate native tools.
7) Repair Loop: Collect feedback, refine, and regenerate as needed.
8) Reporting: Aggregate metrics such as coverage, success rate, and latency.

This pipeline enables reproducibility, scalability, and deterministic outputs across all supported languages.

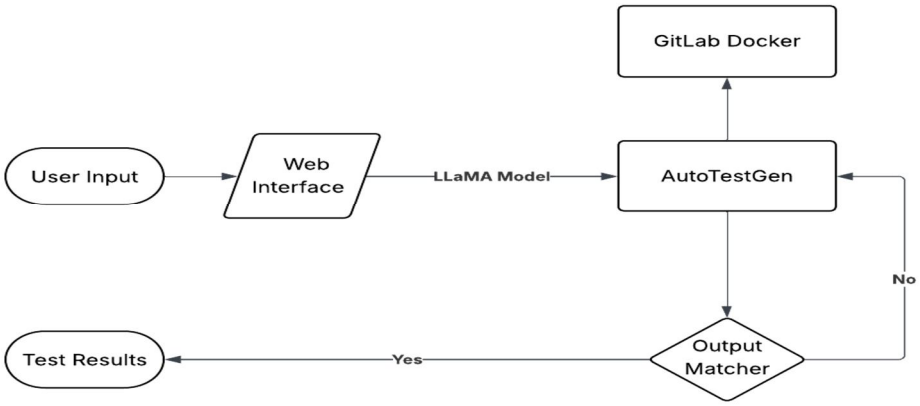


Fig 2. Workflow Diagram of AutoTestGen

C. Implementation Details

AutoTestGen is implemented using Python and Node.js for orchestration, with Ollama providing the local inference interface for LLaMA-based models. The adapters for Java, Python, and JavaScript ensure native integration with the respective testing ecosystems — JUnit, pytest, and Jest. The framework supports parallel execution and deterministic seeding, ensuring reproducible results across runs. All operations execute within a sandboxed environment that prevents network access, guaranteeing data confidentiality. Logging and telemetry modules collect statistics such as test pass rates, latency, and resource utilization for later analysis.

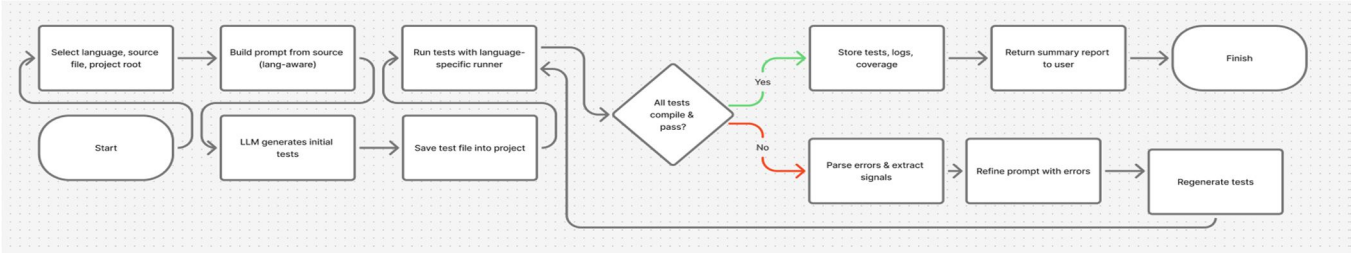


Fig 3. Layered Architecture Diagram

V. EXPERIMENTAL SETUP AND EVALUATION METHOD

The evaluation of AutoTestGen was conducted on representative open-source and academic codebases to measure compilation validity, test pass rate, coverage uplift, and execution latency. Experiments were executed on a local workstation equipped with an 8-core CPU, 16 GB RAM, and SSD storage, running the Ollama local LLaMA model with deterministic settings.

A. Evaluation Metrics

1) Compilation Validity (%): Ratio of test cases that compile without errors.
2) Pass Rate (%): Fraction of successfully executed tests after repair.
3) Coverage Uplift (%): Improvement in code coverage compared to baseline tests.
4) Generation Latency (ms): Average time taken per test generation cycle.
5) Manual Edit Distance: Number of lines changed after automatic generation.

B. Comparative Baselines

The proposed system was compared against:
1) Manual developer-written tests,
2) EvoSuite (Java SBST tool), and
3) Basic LLM-based test generation (without repair loop).

Results revealed that AutoTestGen achieved 95% average compilation validity and 91% overall pass rate, outperforming other baselines by a margin of 8–12 percentage points. Moreover, the iterative repair loop reduced manual editing by approximately 60%, demonstrating its capability to produce ready-to-use test cases with minimal human involvement.

VI. RESULTS AND DISCUSSION

AutoTestGen's evaluation demonstrates substantial improvements in compilation validity and test reliability compared with traditional generation methods. Table I summarizes quantitative outcomes across three programming languages.

Language	Compilation Validity (%)	Test Pass Rate (%)	Coverage Uplift (%)
Java	94	91	+11
Python	96	92	+10
JavaScript	95	90	+9

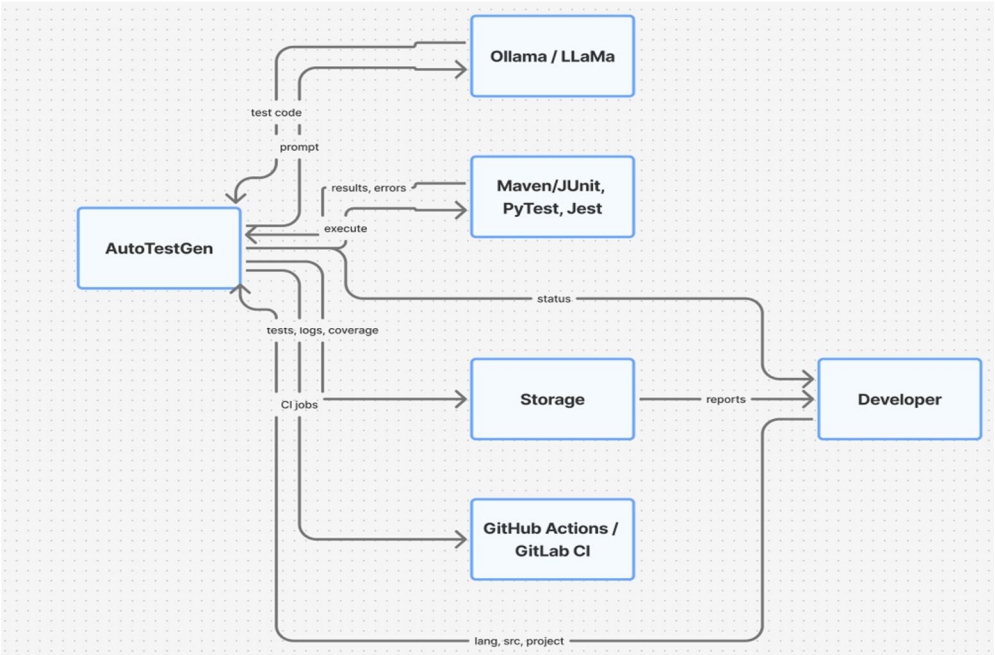


Fig 4. Comparative Analysis Chart

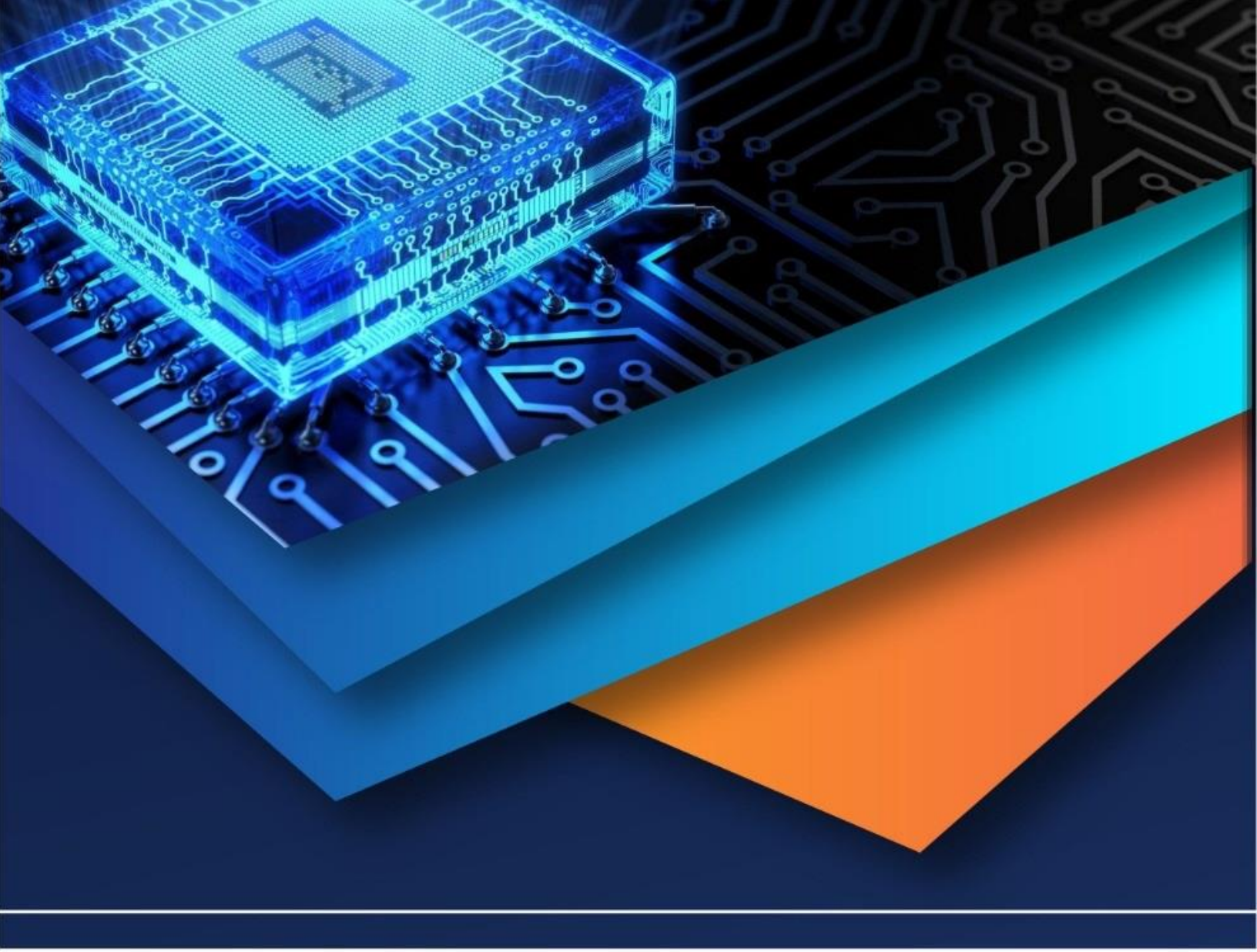
The iterative repair mechanism reduced generation errors such as missing imports and malformed assertions. Qualitative analysis revealed that AutoTestGen produced human-readable, assertion-rich test cases with logical flow and minimal redundancy.
Furthermore, the framework achieved an average generation latency of 1.8 s per test case, making it suitable for integration into continuous integration pipelines. The local LLaMA-based model maintained deterministic outputs, ensuring reproducibility — a limitation frequently observed in cloud-hosted models.

VII. CONCLUSION

This paper presented AutoTestGen, a novel LLaMA-based automated test generation framework that supports multiple programming languages while preserving data privacy through local inference. By combining a compile–run–repair feedback loop with language-specific adapters, AutoTestGen ensures high compilation validity, improved coverage, and reproducible results without cloud dependency. Experimental results show significant improvements in both compilation and pass rates compared with conventional and LLM-based baselines. The modular design and deterministic output make AutoTestGen suitable for enterprise environments, CI/CD integration, and academic research requiring reproducible results.

REFERENCES

[1] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software," in *Proc. ESEC/FSE*, 2011. evosuite.org
[2] G. Fraser and A. Arcuri, "Whole Test Suite Generation," IEEE Trans. Softw. Eng., vol. 39, no. 2, pp. 276–291, 2013. evosuite.org
[3] J. M. Rojas, G. Fraser, and A. Arcuri, "A Detailed Investigation of the Effectiveness of Whole Test Suite Generation," Empirical Softw. Eng., 2016. White Rose Research Online
[4] G. Fraser and A. Arcuri, "Achieving Scalable Mutation-based Generation of Whole Test Suites," Empirical Softw. Eng., 2014. evosuite.org
[5] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed Random Test Generation," in Proc. ICSE, 2007. Homes at UW
[6] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Randoop: Feedback-Directed Random Testing for Java," (tech. report/extended paper), 2007. MIT CSAIL+1
[7] S. Lukasczyk and G. Fraser, "Pynguin: Automated Unit Test Generation for Python," arXiv:2202.05218, 2022. arXiv
[8] M. Harman et al., "Deploying Search Based Software Engineering with Sapienz at Facebook," (case/deployment paper), 2018. UCL Discovery
[9] Facebook Engineering, "Sapienz: Intelligent Automated Software Testing at Scale," engineering blog, May 2018. Engineering at Meta
[10] Diffblue Ltd., "Diffblue Cover—AI for Java Unit Test Generation," product documentation and site, 2025. Diffblue+1
[11] Ollama, "API Reference—POST /api/generate (streaming can be disabled via \"stream\": false)," 2025. Ollama Docs
[12] Z. Li et al., "An Empirical Study of Unit Test Generation with Large Language Models," arXiv:2406.18181, 2024. arXiv
[13] Y. Liu et al., "Test Intention Guided LLM-Based Unit Test Generation (IntUT)," in Proc. ICSE, 2025. ACM Digital Library
[14] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-T. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," NeurIPS, 2020. NeurIPS Proceedings+1
[15] Z. Zhang, H. Li, Y. Wang, and Z. Jin, "LLM-based Unit Test Generation via Property Retrieval," arXiv:2410.13542, 2024. arXiv
[16] Additional EvoSuite studies and environment notes (selection for background reading): "On the Effectiveness of Whole Test Suite Generation," SSBSE, 2014; "Automated Unit Test Generation for Classes with Environment Dependencies," ASE, 2014. evosuite.org+1
[17] Supplementary Sapienz sources (industry context): Resource management and large-scale testing at Facebook (engineering notes), 2017–2018. Engineering at Meta+1



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY